从初学到精通系列

Java 从初学到精通

辛立伟　张　帆　等编著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

Java 不但是最为流行的计算机语言，而且是优秀的纯面向对象语言。本书针对初学者和自学者的特点，以通俗易懂的语言介绍 Java 语言，力求做到深入浅出，将复杂的概念用简洁浅显的语言娓娓道来。

全书内容分为 5 篇，共计 27 章，涵盖了 Java 程序设计语言基础、Java 面向对象基础、Java 图形化界面编程技术、Java 网络应用程序开发技术、Java 数据库应用程序开发技术等方面的内容。为了便于读者学习和实际应用，本书最后一篇通过对几个完整项目案例的分析，详细讲解了使用 Java 语言开发游戏程序、网络程序和数据库应用程序的完整过程，有利于初学者锻炼使用 Java 语言解决实际问题的能力。

本书提供有大量翔实的编程实例和丰富的项目案例，通过对项目的实现和讲解，以提高读者解决实际问题的能力，使读者逐步具备利用 Java 语言来开发应用程序的能力，特别适合于想学习 Java 语言的初学者和大中专院校学生阅读，也可作为有一定 Java 基础的软件开发人员和其他有关人员的参考资料。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Java 从初学到精通 / 辛立伟等编著. —北京：电子工业出版社，2010.6
（从初学到精通系列）
ISBN 978-7-121-10648-4

Ⅰ. ①J… Ⅱ. ①辛… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2010）第 058032 号

责任编辑：高洪霞
印　　刷：北京东光印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：38.75　字数：1013 千字
印　　次：2010 年 6 月第 1 次印刷
印　　数：4000 册　　定价：69.00 元（含 DVD 光盘 1 张）

凡所购买电子工业出版社图书如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

从此初学，走向精通

The book is the nourishment in the whole world.
Life the inside have no book, seem to have no sunlight;
The inside of the intelligence has no book, seem the bird have no wing.

书籍是全世界的营养品。生活里没有书籍，
就好像没有阳光；智慧里没有书籍，就好像鸟儿没有翅膀。

——莎士比亚



为什么要写这样一本书

Java 语言自 1995 年诞生以来，经过十多年的发展和应用，技术日益成熟，现已有 Java SE、Java EE 和 Java ME 三大平台。Java 已成为当今网络计算平台的标准，也成为全球程序员的首选开发平台。今天，Java 技术已无处不在，小至信用卡，大到企业级应用系统、美国宇航系统等。到 2008 年，全球已有超过 15 亿台手机和手持设备应用 Java 技术。Java 技术正显示出高速的增长力，Java 技术所驱动的业务每年都在 1000 亿美元以上。同时，Java 技术以其跨平台特性和良好的可移植性，以及平缓的学习曲线，成为广大软件开发技术人员的挚爱。

为了方便初学者学习 Java 语言，笔者花费半年多时间写作这本书。本书全面地介绍了 Java 语言基础，以实例介绍了 Java 语言的编程技术，以实际项目案例讲解了 Java 应用程序的开发过程。学完本书之后，力求让读者既掌握到 Java 语言的精髓，又有项目实践的本领。

本书的特点

1．细致体贴的讲解

本书假定读者没有任何的编程技能基础，讲解时尽可能地站在读者的角度，循序渐进，由浅入深，用准确的语言总结概念➠用直观的图示演示过程➠用详细的注释解释代码➠用形象的比方帮助记忆。效果如下：

① **知识点介绍**　全面、准确、清晰是其显著特点，一般放在每一节开始位置，让零基础的读者了解相关概念，顺利入门。

② **范例**　本书精心选择的编程示例，可实践性强，注意让读者能从实例运用中举一反三，既有利于初学者对基本知识点的理解和掌握，又锻炼了读者的编程能力。

③ **范例代码**　与范例编号对应，层次清楚、语句简洁、注释丰富，体现了代码优美的原则，有利于读者养成良好的代码编写习惯。读者可以直接根据这些源程序快速编写程序，直接切入相关应用。

④ **运行结果**　对范例给出运行结果和对应图示，帮助读者更直观地理解范例代码、验证结果。

⑤ **代码解析**　本书对范例代码中的关键代码配以图示并进行详细解释，有助于读者掌握相关概念和知识。

⑥ **贴心的提示**　为了便于读者阅读，全书还穿插着一些技巧、提示等小贴士，体例约定如下：

- 提示：通常是一些贴心的提醒，让读者加深印象或提供建议，或者解决问题的方法。
- 注意：提出学习过程中需要特别注意的一些知识点和内容，或者相关信息。
- 警告：对操作不当或理解偏差将会造成的灾难性后果做警示，以加深读者印象。

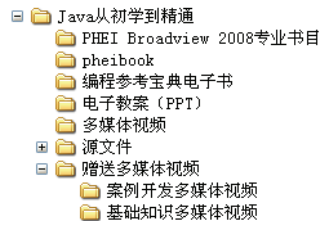
⑦ **小结**　每章最后提供对本章所介绍知识的简单总结，有利于读者加深印象，巩固总结。

经作者多年的培训和授课证明，以上讲解方式是最适合读者学习的方式，读者按照这种方式，会非常轻松、顺利地掌握本书知识。



2．实用超值的 DVD 光盘

为了帮助读者比较直观地学习，本书附赠 DVD 光盘，内容包括多媒体视频、电子教案（PPT）、编程参考宝典电子书、各章习题答案和实例源代码等。

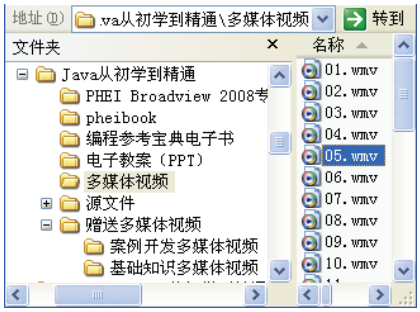


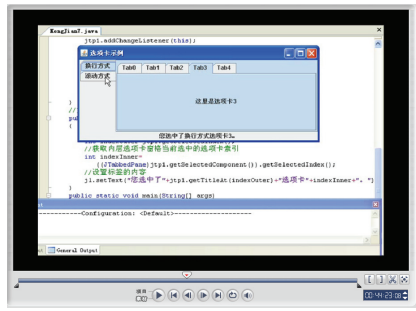
- **多媒体视频**

配有长达 12 小时手把手教学视频，讲解关键知识点界面操作和书中的一些综合练习题。作者亲自配音、演示，手把手教会读者使用。此外，还随书免费赠送 41 小时相关知识多媒体视频，帮助读者巩固和加深对所学内容的理解和掌握。

- **电子教案（PPT）**

本书可以作为高校相关课程的教材或课外辅导书，所以笔者特别为本书制作了电子教案（PPT），以方便老师教学使用。

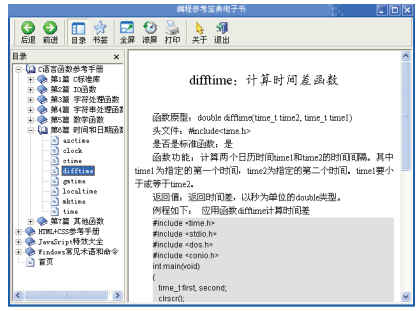




- 编程参考宝典电子书

为方便广大读者学习，特别制作了编程开发参考电子书，供读者查阅和参考。





3．提供完善的技术支持

本书提供了论坛：http://www.rzchina.net，读者可以在上面提问交流。另外，论坛上还有一些小的教程、视频动画和各种技术文章，可帮助读者提高开发水平。

4．丰富的额外素材下载

相关的开发素材文件，在 www.broadview.com.cn 提供下载。

本书适合哪些读者阅读

- 想从事软件开发的入门者
- 有一点经验的开发人员
- 从其他语言迁移过来的开发人员
- 大中专院校的学生
- 社会培训学生

本书作者

本书由辛立伟、张帆主笔编写，其他参与编写的人员有昊燃、方振宇、陈冠佐、傅奎、陈勤、梁洋洋、毕梦飞、陈庆、柴相花、陈非凡、陈华、陈嵩、承卓。

编者

目　　录

第 1 篇　Java 语言基础篇

第 1 章　Java 语言概述……30

本章介绍了 Java 语言的背景、特点、开发环境、开发过程及开发工具的使用。在学习本章内容时，要重点理解 Java 技术的两个含义：Java 程序设计语言与 Java 平台。重点掌握如何搭建 Java 程序的开发环境，包括下载及安装 Java 开发工具包 JDK、配置环境变量及 JDK 环境测试等。

1.1　Java 语言简介……30

1.1.1　Java 语言的起源……30

1.1.2　使用 Java 语言的优点……31

1.2　Java 语言与 Java 平台……32

1.2.1　Java 程序设计语言……32

1.2.2　Java 平台……32

1.3　搭建 Java 程序开发环境……33

1.3.1　系统要求……33

1.3.2　下载 Java 程序开发工具包 JDK……34

1.3.3　安装 JDK……35

1.3.4　在 Windows 系统下配置 JDK……36

1.4　开发第一个 Java 应用程序……37

1.4.1　创建第一个 Java 应用程序源文件……37

1.4.2　将 HelloWorldApp.java 源文件编译为.class 文件……38

1.4.3　运行 HelloWorldApp 应用程序……39

1.4.4　Java 应用程序的基本结构……40

1.5　小结……41

第 2 章　Java 语言基本语法……42

本章深入学习了 Java 语言的基本语法，重点对常量与变量及数据类型和运算符做了详细的讲述。在本章的学习过程中，可能对 Java 数据类型的转换难以理解，因此读者一定要多做练习，通过上机练习体会和掌握 Java 的基本语法。

2.1　标识符和关键字……42

2.1.1　标识符……42

2.1.2　关键字……42

2.1.3　标识符命名规则……43

2.2 常量与变量 43
2.2.1 常量的概念及常量声明 43
2.2.2 变量的概念及变量声明 44
2.3 基本数据类型 45
2.3.1 数据类型 45
2.3.2 整数类型 45
2.3.3 浮点类型 47
2.3.4 字符类型 47
2.3.5 逻辑类型 49
2.3.6 不同数据类型间的转换 49
2.4 运算符与表达式 51
2.4.1 算术运算符与算术表达式 51
2.4.2 赋值运算符与赋值表达式 52
2.4.3 自增和自减运算符 53
2.4.4 关系运算符和关系表达式 54
2.4.5 逻辑运算符和逻辑表达式 54
2.4.6 位运算符 57
2.4.7 三元运算符 59
2.4.8 运算符的优先级 60
2.5 小结 60
第 3 章 数组 61

本章深入学习了 Java 语言的数组，了解了 Java 语言中数组的概念，学习了如何声明、创建、访问和修改数组。本章重点是一维数组和二维数组。其中对于数组的创建及二维数组的创建和使用是本章学习的难点，建议读者多上机练习。

3.1 数组的概念与特点 61
3.1.1 什么是数组 61
3.1.2 Java 语言中数组的特点 62
3.2 一维数组 62
3.2.1 声明一维数组变量 62
3.2.2 创建一维数组对象 63
3.2.3 访问一维数组元素 63
3.2.4 修改一维数组元素 64
3.3 多维数组 65
3.3.1 声明二维数组变量 65
3.3.2 创建二维数组对象 66
3.3.3 访问二维数组元素 67

3.3.4 修改二维数组元素 ····· 67
3.4 小结 ····· 68

第 4 章 程序流程控制语句 ····· 69

本章深入介绍了有关程序流程控制的语句，主要包括选择语句、循环语句和跳转语句。其中的 while 循环语句因其灵活方便，读者应重点掌握。本章最后讲到的跳转语句主要用于提高循环语句的灵活性，特别是标签 break 语句，是本章掌握的难点。

4.1 选择语句 ····· 69
4.1.1 if 选择语句 ····· 69
4.1.2 if-else 双分支选择语句 ····· 70
4.1.3 if-else-if 多分支选择语句 ····· 71
4.1.4 选择语句的嵌套 ····· 74
4.1.5 switch 多分支开关语句 ····· 76
4.1.6 if 选择语句与 switch 开关语句的区别 ····· 78
4.2 循环语句 ····· 78
4.2.1 while 循环语句 ····· 78
4.2.2 do-while 循环语句 ····· 80
4.2.3 for 循环语句 ····· 83
4.2.4 循环语句的嵌套 ····· 85
4.2.5 使用增强型 for 语句遍历一维数组中的元素 ····· 87
4.2.6 对一维数组进行排序 ····· 88
4.2.7 使用增强型 for 语句遍历二维数组中的元素 ····· 89
4.3 跳转语句 ····· 90
4.3.1 break 跳转语句 ····· 90
4.3.2 continue 跳转语句 ····· 93
4.4 实例 ····· 94
4.4.1 实例 1：译密码 ····· 95
4.4.2 实例 2：打印九九乘法表 ····· 96
4.5 小结 ····· 97

第 5 章 数字、字符和字符串处理 ····· 98

本章深入介绍了 Java 语言中有关数字、字符和字符串的处理的相关知识，重点讲解了 java.lang 包中用来处理数字的 Number 及其子类、专门用来处理数学运算的 Math 类、专门用来处理字符的 Character 类，以及专门用来创建和操作字符串的类 String。

5.1 数字 ····· 98
5.1.1 Numbers 数字包装器类 ····· 98
5.1.2 数字的格式化输出 ····· 100

5.1.3 Math 数学运算处理类 …… 104
5.2 字符 …… 107
5.2.1 Character 字符包装器类 …… 107
5.2.2 转义字符序列 …… 108
5.3 使用 String 类 …… 109
5.3.1 创建字符串对象 …… 109
5.3.2 获取字符串的长度 …… 110
5.3.3 连接字符串 …… 110
5.3.4 字符串与字符数组 …… 112
5.3.5 在一个字符串中操纵字符 …… 113
5.3.6 字符串比较和字符串部分内容比较 …… 117
5.4 使用 StringBuilder 类 …… 120
5.4.1 认识 StringBuilder 类 …… 120
5.4.2 StringBuilder 对象的长度和容量 …… 121
5.4.3 StringBuilder 类提供的操作方法 …… 121
5.5 实例 …… 123
5.5.1 实例 1：用户登录验证程序 …… 123
5.5.2 实例 2：敏感词过滤程序 …… 125
5.6 小结 …… 126

第 2 篇 Java 面向对象篇

第 6 章 类和对象 …… 128

本章主要学习了类和对象的相关概念，包括类的声明、类的实例（即对象）。本章所讲的知识是 Java 面向对象语言中最基本的概念，所涉及的类和声明、对象的构造、访问控制修饰符都是读者学习的重点。其中的构造器的工作过程、访问控制和 this 关键字的使用是本章学习的难点。

6.1 面向对象简介 …… 128
6.1.1 什么是对象 …… 128
6.1.2 什么是类 …… 130
6.1.3 什么是继承 …… 131
6.1.4 什么是接口 …… 131
6.1.5 什么是包 …… 132
6.2 定义类 …… 132
6.2.1 类的基本结构 …… 132
6.2.2 声明类的成员变量 …… 134
6.2.3 声明类的成员方法 …… 135
6.2.4 成员方法命名 …… 136
6.2.5 方法重载 …… 137

6.2.6 调用成员方法 137
6.2.7 为类提供构造器 139
6.3 向方法或构造器内传递信息 140
6.3.1 使用参数 140
6.3.2 实参与形参的关系 141
6.3.3 参数类型 143
6.3.4 传递任意数量的参数 143
6.3.5 参数名称 144
6.3.6 传递原始数据类型参数 144
6.3.7 传递引用数据类型参数 145
6.4 对象 145
6.4.1 对象实例 145
6.4.2 创建对象 147
6.4.3 使用对象 150
6.5 方法的返回值 152
6.5.1 返回值 152
6.5.2 返回类或接口 153
6.6 使用 this 关键字 153
6.6.1 对字段使用 this 154
6.6.2 对构造器使用 this 154
6.7 控制对类的成员的访问 155
6.8 理解实例成员和类成员 156
6.8.1 类变量 156
6.8.2 类方法 158
6.8.3 常量 158
6.8.4 实例：AutoCar 类 158
6.9 初始化字段 159
6.9.1 静态初始化块 159
6.9.2 初始化实例成员 160
6.10 枚举类型 161
6.11 标注 164
6.11.1 标注的用法 164
6.11.2 文档标注 164
6.11.3 预定义标注 165
6.11.4 标注处理 168
6.12 小结 168

第 7 章　继承 …… 169

本章主要学习了类的继承。这是面向对象语言的三大特征之一。本章学习的重点是继承中方法的覆盖和方法的重写、super 关键字的使用及构造器链。读者在学习本章内容时对 super 关键字的使用及构造器链的调用要细心体会，这是本意学习的难点。

7.1　继承概述 …… 169
7.1.1　继承的概念 …… 169
7.1.2　Java 平台中的类层次 …… 169
7.1.3　一个继承的示例 …… 170
7.1.4　在子类中可以做的事情 …… 171
7.2　从子类中访问父类的私有成员 …… 172
7.3　对象类型转换 …… 173
7.3.1　隐式对象类型转换 …… 173
7.3.2　强制对象类型转换 …… 174
7.3.3　使用 instanceof 运算符 …… 174
7.4　覆盖和隐藏超类中的方法 …… 174
7.4.1　覆盖超类中的实例方法 …… 175
7.4.2　协变覆盖 …… 175
7.4.3　隐藏超类中的类方法 …… 176
7.4.4　方法覆盖和方法隐藏后的修饰符 …… 178
7.4.5　方法覆盖和方法隐藏总结 …… 178
7.5　隐藏超类中的字段 …… 179
7.6　使用 super 关键字访问超类中的成员 …… 180
7.6.1　调用超类中被覆盖的方法 …… 180
7.6.2　访问超类中被隐藏的字段 …… 180
7.7　使用 super 关键字调用超类的构造器 …… 181
7.7.1　调用超类的无参构造器 …… 181
7.7.2　调用超类的带参构造器 …… 183
7.7.3　构造器链 …… 184
7.8　Object 类 …… 185
7.8.1　作为超类的 Object 类 …… 185
7.8.2　Object 类的对象克隆方法 clone() …… 186
7.8.3　Object 类的对象比较方法 equals() …… 186
7.8.4　Object 类的对象清除方法 finalize() …… 187
7.8.5　Object 类的获取对象类信息的方法 getClass() …… 187
7.8.6　Object 类的获得对象内存地址的方法 hashCode() …… 188
7.8.7　Object 类的对象字符串表示方法 toString() …… 188
7.9　编写 final 类和 final 方法 …… 188
7.9.1　使用 final 关键字声明 final 方法 …… 188

7.9.2 使用 final 关键字声明 final 类……189
7.10 小结……189
第 8 章 接口……190
本章主要介绍了接口的概念、接口的声明及接口的实现。在学习本章内容时，读者重点理解接口的概念和学习如何实现接口。在将接口作为类型使用上，读者可能会感到难以理解，需用心体会。
8.1 理解接口……190
8.1.1 一般接口的概念……190
8.1.2 Java 中的接口……191
8.1.3 在 Java 中定义接口……191
8.1.4 在 Java 中使用接口……191
8.1.5 作为 API 的接口……192
8.1.6 接口和多继承……192
8.2 定义接口……193
8.2.1 接口声明……193
8.2.2 接口体……193
8.3 实现接口……194
8.3.1 实现接口的语法……194
8.3.2 接口实例：Relatable……195
8.3.3 实现接口 Relatable……195
8.3.4 编译并运行带有接口的实例……196
8.3.5 实现多个接口时的常量和方法冲突问题……197
8.4 将接口作为类型使用……198
8.5 改写接口……199
8.6 小结……200
第 9 章 抽象类与嵌套类……201
本章介绍了两种特殊的类型：抽象类和嵌套类。抽象类是实现多态的一种途径，嵌套类主要应用在事件监听器的构建上。其中抽象类和接口的异同是读者学习的一个重点和难点。嵌套类的定义也是学习中容易迷惑的地方。建议读者在学习到这部分内容时，如果暂时不能理解的话，不妨先放一放。等学过后面的事件监听机制以后，再回来学习这部分内容，相信就容易理解了。
9.1 抽象类和抽象方法……201
9.1.1 抽象方法……201
9.1.2 抽象类……201
9.1.3 抽象类与接口对比……202
9.1.4 抽象类实例……202
9.1.5 抽象类实现接口……203

9.1.6 抽象类的类成员 204
9.2 嵌套类 204
9.2.1 为什么要使用嵌套类 205
9.2.2 静态嵌套类 205
9.2.3 内部类 205
9.2.4 内部类实例 206
9.2.5 局部内部类和匿名内部类 207
9.2.6 内部类的修饰符 207
9.3 小结 207
第 10 章 泛型 208

本章主要描述了什么是泛型、为什么要使用泛型及如何使用泛型。读者在学习本章时，重点理解什么是泛型及为什么要使用泛型。读者可能在学习本章的过程中对如何使用泛型还存在迷惑，没有关系，完全可以先不去理会它。在具体使用泛型最多的地方一般是在 Java 的集合框架部分，所以读者可以在学习 Java 集合框架部分内容时，再回来结合本章的内容来掌握泛型的使用。

10.1 泛型简介 208
10.1.1 为什么要使用泛型 208
10.1.2 一个简单的 Box 类 208
10.2 泛型类型 210
10.2.1 Box 类的泛型版本 210
10.2.2 类型参数命名惯例 212
10.3 泛型方法和泛型构造器 212
10.4 限定的类型参数 213
10.5 泛型子类型 214
10.6 使用通配符 215
10.7 类型擦除 216
10.8 小结 217
第 11 章 包 218

本章主要介绍了 Java 语言中包的概念及如何创建和使用包。包是 Java 组织资源的一种方式，读者学习起来应该不会感到吃力。只是如何将包对应到具体的物理路径上，读者应该细心体会。因为 Java 平台的许多实现依赖于分组的文件系统来管理源文件和 class 文件的，包名往往对应着系统中实际的目录名，这一点读者一定要注意。

11.1 使用包 218
11.2 创建包 219
11.2.1 创建包的语句 219
11.2.2 命名包 220

11.2.3 包命名惯例 220
11.3 使用包中的成员 221
11.3.1 通过全限定名引用包成员 221
11.3.2 导入包中的成员 221
11.3.3 导入整个包 222
11.3.4 包的层级 222
11.3.5 包成员名称冲突 223
11.3.6 静态导入语句 223
11.4 管理源文件和 class 类文件 224
11.4.1 管理实现策略 224
11.4.2 设置 CLASSPATH 系统变量 225
11.5 小结 225
第 12 章 集合 226

本章主要学习了 Java 集合框架，包括接口、实现和算法。其中核心集合接口是 Java 集合框架的基础。实现是用于存储集合的数据对象，是实现了集合接口的类。

12.1 Java 集合框架概述 226
12.1.1 Java 集合框架 226
12.1.2 使用 Java 集合框架的益处 227
12.2 核心集合接口 227
12.3 Collection 接口 229
12.3.1 转换构造器 229
12.3.2 Collection 接口的定义 229
12.3.3 Collection 接口的基本操作 230
12.3.4 遍历 Collection 接口 230
12.3.5 Collection 接口的批量操作 231
12.3.6 Collection 接口的数组操作 231
12.4 Set 接口 232
12.4.1 Set 接口的定义 232
12.4.2 Set 接口的基本操作 233
12.4.3 Set 接口的批量操作 234
12.4.4 Set 接口的数组操作 235
12.5 List 接口 235
12.5.1 List 接口的定义 235
12.5.2 List 接口与 Vector 的比较 236
12.5.3 从 Collection 继承的操作 236
12.5.4 按位置访问和查找操作 237

12.5.5 List 迭代器 238
12.5.6 子集合操作 240
12.6 Queue 接口 242
12.6.1 Queue 接口的定义 242
12.6.2 Queue 接口的操作 242
12.6.3 Queue 接口应用 243
12.7 Map 接口 243
12.7.1 Map 接口的定义 243
12.7.2 Map 接口的基本操作 244
12.7.3 Map 接口的批量操作 245
12.7.4 集合视图 245
12.8 SortedSet 接口 246
12.8.1 SortedSet 接口的定义 246
12.8.2 SortedSet 接口的操作 247
12.8.3 SortedSet 接口的标准构造器 247
12.8.4 子集合操作 247
12.8.5 端点操作 248
12.9 SortedMap 接口 248
12.9.1 SortedMap 接口的定义 249
12.9.2 SortedMap 接口的操作 249
12.9.3 SortedMap 接口的标准构造器 249
12.9.4 与 SortedSet 接口的比较 249
12.10 实现 250
12.10.1 实现的类型 250
12.10.2 Set 接口的实现 251
12.10.3 List 接口的实现 252
12.10.4 Map 接口的实现 252
12.10.5 Queue 接口的实现 253
12.11 算法 253
12.11.1 Collections 类 254
12.11.2 排序 254
12.11.3 乱序 254
12.11.4 常规数据处理 255
12.11.5 查找 255
12.11.6 组成 255
12.11.7 查找极值 255
12.12 小结 256

第 3 篇　Java 图形图像篇

第 13 章　创建一个图形界面应用程序 ····· 258

本章学习如何使用 Swing 工具包创建带图形界面的 Java 应用程序。在学习本章时，重点理解顶层窗口的概念，特别是内容面板。因为 Java 应用程序中要求其他组件都要位于内容面板之上。在学习窗体及窗体事件响应时可能会有些理解上的困难，不用担心，只要了解其工作原理即可，在实际开发中，集成开发工具会给程序员完成很多基础性的工作的。

13.1　Java Swing 简介 ····· 258
13.1.1　什么是 Swing ····· 258
13.1.2　Swing 特性 ····· 259
13.2　创建一个简单的图形界面程序 ····· 260
13.2.1　HelloWorldSwing.java 界面程序示例 ····· 260
13.2.2　编译并运行 HelloWorldSwing 程序 ····· 261
13.3　使用 Swing 顶层容器 ····· 262
13.3.1　Swing 中的顶层容器类 ····· 262
13.3.2　顶层容器和容器层级 ····· 264
13.3.3　向内容面板中添加组件 ····· 264
13.3.4　给窗体添加一个菜单栏 ····· 265
13.3.5　根面板 ····· 265
13.4　使用 JFrame 类创建图形界面程序窗体 ····· 265
13.4.1　创建并显示一个窗体 ····· 265
13.4.2　创建窗体的一般步骤 ····· 266
13.4.3　指定窗口装饰 ····· 267
13.4.4　响应窗口关闭事件 ····· 269
13.4.5　窗体 API ····· 270
13.5　小结 ····· 271

第 14 章　使用 Swing 基本组件 ····· 272

本章主要介绍 JComponent 类和 Swing 的基本组件。JComponent 类是所有 Swing 中非顶层组件的直接或间接超类，它定义了公共的 API，所以了解 JComponent 类有助于掌握其他 Swing 组件的使用。

14.1　认识 JComponent 类 ····· 272
14.1.1　JComponent 特性 ····· 272
14.1.2　Jcomponent API ····· 273
14.2　基本控件 ····· 276
14.2.1　按钮组件 JButton ····· 276
14.2.2　复选框组件 JCheckBox ····· 280
14.2.3　单选按钮组件 JRadioButton ····· 281
14.2.4　文本框组件 JTextField ····· 282

14.2.5 密码框组件 JPasswordField …… 284
14.2.6 组合框组件 JComboBox …… 286
14.2.7 滑块组件 JSlider …… 288
14.2.8 微调控制组件 JSpinner …… 290
14.2.9 菜单组件 JMenu …… 292
14.2.10 列表框组件 JList …… 297
14.3 不可编辑的信息显示控件 …… 300
14.3.1 标签组件 JLabel …… 300
14.3.2 进度条组件 JProgressBar …… 302
14.3.3 分隔线组件 JSeparator …… 303
14.3.4 工具提示组件 JToolTip …… 305
14.4 小结 …… 305

第 15 章 使用 Swing 高级组件 …… 306

本章主要讲解 Swing 的高级组件，包括交互式显示高度格式化信息控件，如颜色选择器 JColorChooser、文件选择器 JFileChooser、表组件 JTable 和树组件 JTree 等。另外，还介绍了非顶层容器控件及特殊用途窗口控件。最后两节介绍了如何在 Swing 组件中使用 HTML 以格式化文本，以及 Swing 提供的各种简单而实用的对话框。

15.1 交互式显示高度格式化信息控件 …… 306
15.1.1 颜色选择器 JColorChooser …… 306
15.1.2 文件选择器 JFileChooser …… 307
15.1.3 文本编辑组件 JEditorPane 和 JTextPane …… 309
15.1.4 文本区组件 JTextArea …… 311
15.1.5 表组件 JTable …… 312
15.1.6 树组件 JTree …… 315
15.2 非顶层容器控件 …… 318
15.2.1 面板组件 JPanel …… 318
15.2.2 滚动面板 JScrollPane …… 320
15.2.3 拆分面板 JSplitPane …… 321
15.2.4 选项卡面板 JTabbedPane …… 323
15.2.5 工具栏 JToolBar …… 326
15.3 特殊用途窗口控件 …… 328
15.3.1 内部窗体组件 JInternalFrame …… 328
15.3.2 层面板 JLayeredPane …… 330
15.3.3 根面板 JRootPane …… 332
15.4 在 Swing 组件中使用 HTML …… 334
15.5 顶层容器对话框 JDialog …… 336
15.5.1 对话框简介 …… 336

15.5.2 使用 JOptionPane 类创建简单对话框 ····· 336
15.6 小结 ····· 339
第 16 章 程序界面布局 ····· 340
本章主要介绍了 Swing 中的程序界面布局方式，包括 BorderLayout、BoxLayout、CardLayout、FlowLayout、GridBagLayout、GridLayout、GroupLayout 和 SpringLayout。
16.1 标准布局管理器简介 ····· 340
16.1.1 BorderLayout 边框布局 ····· 340
16.1.2 BoxLayout 布局 ····· 341
16.1.3 CardLayout 卡片布局 ····· 341
16.1.4 FlowLayout 流动布局 ····· 341
16.1.5 GridBagLayout 网格包布局 ····· 341
16.1.6 GridLayout 网格布局 ····· 341
16.1.7 GroupLayout 布局 ····· 342
16.1.8 SpringLayout 布局 ····· 342
16.2 在程序中使用布局管理器 ····· 342
16.2.1 设置布局管理器 ····· 342
16.2.2 向容器中添加组件 ····· 342
16.2.3 提供组件大小和排列策略 ····· 343
16.2.4 设置组件之间的间隙 ····· 343
16.2.5 设置容器的语言方向 ····· 343
16.2.6 选择布局管理器 ····· 344
16.3 使用各种布局管理器 ····· 344
16.3.1 使用 BorderLayout ····· 345
16.3.2 使用 BoxLayout ····· 347
16.3.3 使用 CardLayout ····· 348
16.3.4 使用 FlowLayout ····· 349
16.3.5 使用 GridLayout ····· 350
16.3.6 使用 GridbagLayout ····· 352
16.4 实例：设计用户登录界面 ····· 358
16.4.1 设计界面布局 ····· 358
16.4.2 代码实现 ····· 358
16.5 小结 ····· 362
第 17 章 事件处理 ····· 363
本章介绍了程序事件处理原理和 Java 中的事件处理机制，并且概括了 Java 中的各种事件类型。在学习本章内容时，重点掌握各种事件类型的处理方法。在了解 Java 的事件处理机制时，可能会有一定的难度。

17.1 事件处理原理 363
17.2 Java 事件处理模型 363
17.3 Java 事件与监听器的类型 364
17.3.1 Java 事件类型 365
17.3.2 Java 监听器类型 366
17.3.3 Java 适配器类 366
17.4 处理动作事件 367
17.4.1 编写动作事件处理程序的步骤 367
17.4.2 处理动作事件的过程 367
17.4.3 处理按钮触发的动作事件 368
17.4.4 处理选择菜单时触发的事件 370
17.4.5 处理文本框触发的事件 373
17.5 处理选项事件 375
17.5.1 编写选项事件处理程序的步骤 375
17.5.2 处理选项事件的过程 375
17.5.3 处理选择项状态改变所触发的事件 376
17.6 处理列表选择事件 378
17.6.1 编写列表事件处理程序的步骤 378
17.6.2 处理列表选择事件的过程 379
17.6.3 处理选择列表项时所触发的事件 379
17.7 处理焦点事件 381
17.7.1 编写焦点事件处理程序的步骤 381
17.7.2 处理焦点事件的过程 382
17.7.3 处理组件获得或失去焦点时所触发的事件 382
17.8 处理键盘事件 384
17.8.1 编写键盘事件处理程序的步骤 384
17.8.2 处理键盘事件的过程 384
17.8.3 处理键盘击键时所触发的事件 385
17.9 处理鼠标事件 387
17.9.1 编写鼠标事件处理程序的步骤 387
17.9.2 处理鼠标事件的过程 387
17.9.3 处理鼠标操作所触发的事件 388
17.10 处理鼠标移动事件 390
17.10.1 编写鼠标移动事件处理程序的步骤 390
17.10.2 处理鼠标移动事件的过程 390
17.10.3 处理鼠标移动时所触发的事件 391
17.11 实例：一个简单的用户登录验证程序 393
17.11.1 程序功能 393

17.11.2 为程序实现动作事件监听器 ····· 394
17.11.3 为按钮注册监听器 ····· 394
17.11.4 实现 ActionListener 接口中的 actionPerformed 方法 ····· 395
17.11.5 编译并运行程序 ····· 396
17.12 小结 ····· 396

第 18 章 图形、文本和图像处理 ····· 397

本章介绍了 Java 中对图形、文本和图像的处理，主要是使用 Java 2D API。在本章中，对图像的处理就是读者学习的重点，因为在实际开发中，相对图形和文本的处理，可能使用 Java 的机会并不是很多。其中文本的处理较为复杂，是读者学习的难点。建议读者能理解本章的内容即可。

18.1 Java2D 概述 ····· 397
18.1.1 Java2D 坐标空间 ····· 397
18.1.2 Java2D 渲染模型 ····· 398
18.2 Java2D 绘图入门 ····· 398
18.2.1 创建一个绘图面板 ····· 398
18.2.2 获取 Graphics2D 对象 ····· 399
18.2.3 简单实例：绘制字符串 ····· 399
18.3 绘制几何图形 ····· 400
18.3.1 基本几何对象 ····· 400
18.3.2 使用 Point2D ····· 401
18.3.3 画直线 ····· 401
18.3.4 画直角矩形 ····· 402
18.3.5 画圆角矩形 ····· 402
18.3.6 画椭圆形 ····· 403
18.3.7 画弧形 ····· 404
18.3.8 画二次曲线 ····· 405
18.3.9 画三次曲线 ····· 405
18.3.10 画任意图形 ····· 406
18.3.11 设置画笔样式 ····· 407
18.3.12 设置填充模式 ····· 409
18.3.13 实例：绘制和填充几何图形 ····· 410
18.4 绘制文本 ····· 413
18.4.1 选择字体 ····· 413
18.4.2 获得本地系统已安装的字体 ····· 414
18.4.3 创建一个新的字体 ····· 415
18.4.4 获取某个字体的信息 ····· 416
18.5 绘制图像 ····· 416
18.5.1 读取/加载图像 ····· 416

18.5.2 绘制图像 …… 417
18.5.3 创建并绘制图像 …… 418
18.5.4 写入/保存图像 …… 418
18.5.5 获取可读写的图像类型 …… 419
18.5.6 实例：读写 jpg 图像 …… 419
18.6 小结 …… 421

第 4 篇 Java 开发技术篇

第 19 章 异常处理 …… 424

本章主要向读者介绍了 Java 中的异常处理机制。异常处理机制是保证 Java 程序正常运行、具有较高安全性的重要手段。对于开发良好的编程习惯是非常重要的。读者在学习本章内容时，把重点放在对 Java 程序中的异常处理上。而在学习到自定义异常时，读者可能会感觉到有些困难。建议将自定义异常作为了解内容即可。

19.1 Java 异常 …… 424
19.1.1 Java 程序的编译错误 …… 424
19.1.2 Java 程序的运行错误 …… 425
19.1.3 Java 程序中的逻辑错误 …… 426
19.1.4 Java 程序的异常处理机制 …… 427
19.1.5 Java 语言中的异常处理类 …… 427
19.2 处理 Java 程序中的异常 …… 428
19.2.1 使用 try-catch 语句保护代码和捕获异常 …… 428
19.2.2 使用多个 catch 子句 …… 428
19.2.3 使用 finally 子句 …… 429
19.2.4 可嵌入的 try 块 …… 430
19.3 抛出异常 …… 431
19.3.1 使用 throws 声明可能会抛出异常的方法 …… 431
19.3.2 使用 throw 抛出异常 …… 432
19.3.3 异常类常用方法 …… 433
19.4 自定义异常 …… 433
19.4.1 创建自定义异常类 …… 433
19.4.2 使用 throw 和 try-catch 语句处理自定义异常 …… 434
19.5 小结 …… 435

第 20 章 输入与输出 …… 436

本章主要介绍了 Java 中的数据流，包括输入流和输出流。流的处理是 Java 中很重要的一项技术。读者在学习本章内容时，重点掌握对字节流的处理和对字符流的处理。Java 中流的体系比较复杂，可能会对读者的理解带来困难。建议读者在学习时重点放在字节流、字符流和缓冲流的处理上，其他的部分作为了解内容即可。

20.1 文件类（File 类）的使用 ····· 436
20.2 流的概念 ····· 438
20.2.1 流的定义 ····· 438
20.2.2 输入流与输出流 ····· 438
20.2.3 字节流与字符流 ····· 439
20.3 字节流 ····· 439
20.3.1 InputStream 类与 OutputStream 类 ····· 439
20.3.2 文件输入流 FileInputStream 类与文件输出流 FileOutputStream 类 ····· 440
20.3.3 缓存输入流 BufferedInputStream 类与缓存输出流 Buffered OutputStream 类 ····· 442
20.4 字符流 ····· 443
20.4.1 Reader 类和 Writer 类 ····· 443
20.4.2 字符文件读写器 FileReader 类和 FileWriter 类 ····· 444
20.4.3 带缓存的字符文件读写器 BufferedReader 类和 BufferedWriter 类 ····· 445
20.4.4 打印流 PrintStream 类和 PrintWriter 类 ····· 447
20.5 实现用户输入 ····· 448
20.5.1 使用 System.in 获取用户输入 ····· 448
20.5.2 使用 Scanner 类获取用户输入 ····· 449
20.5.3 使用命令行参数 ····· 449
20.6 小结 ····· 450
第 21 章 并发程序设计 ····· 451

本章介绍了 Java 线程的一些基本知识和简单应用，并进一步讲解了并发程序中几个重要的概念：同步、死锁及高级并发对象。本章的内容对于初学者来说有一定的难度，特别是同步、死锁、不可改变对象及高级并发对象等。建议读者把学习的重点放在线程的理解及两种线程的实现方法上。

21.1 进程和线程 ····· 451
21.1.1 进程 ····· 451
21.1.2 线程 ····· 452
21.2 线程对象 ····· 452
21.2.1 定义并启动一个线程 ····· 452
21.2.2 使用 Sleep 暂停线程执行 ····· 453
21.2.3 中断线程 ····· 454
21.2.4 join 方法 ····· 455
21.2.5 实例：SimpleThreadsDemo.java ····· 456
21.3 同步 ····· 459
21.3.1 线程冲突 ····· 459
21.3.2 内存一致性错误 ····· 460
21.3.3 同步方法 ····· 460

21.3.4 固定锁和同步 461
21.3.5 原子访问 462
21.4 活跃性 463
21.4.1 死锁 463
21.4.2 饥饿和活锁 464
21.5 守护块 464
21.6 不可变对象 467
21.6.1 同步类实例 468
21.6.2 定义不可变对象的策略 469
21.7 高级并发对象 470
21.7.1 锁对象 471
21.7.2 执行器 473
21.7.3 并发集合 474
21.7.4 原子变量 476
21.8 实例：模拟银行系统进行存取款处理 477
21.9 小结 479
第 22 章 网络编程 480

在本章中，学习了如何使用 Java 的 java.net 包所提供的类来进行 Socket 网络编程。重点讲了基于 TCP 协议的可靠的和面向连接的网络通信，以及基于 UDP 协议的不可靠的和无连接的网络通信。这是本章学习的重点。其中应用多线程处理多客户端的访问是本章学习的难点，读者应结合上一章线程部分的内容来理解这部分内容。

22.1 网络编程基础 480
22.1.1 TCP 协议 480
22.1.2 IP 协议 480
22.1.3 UDP 协议 480
22.1.4 端口 481
22.1.5 套接字 481
22.1.6 java.net 包 481
22.2 使用 InetAddress 类封装 IP 地址 481
22.2.1 InetAddress 类 482
22.2.2 实例：获取本地主机和 Internet 公共域名的名称和 IP 地址 482
22.3 基于 URL 的网络编程 483
22.3.1 URL 类 484
22.3.2 直接读取 URL 内容 486
22.3.3 URLConnection 类 487
22.4 基于 TCP 的网络编程 488
22.4.1 Socket 通信模型 488

22.4.2 服务器端程序设计 ····· 489
22.4.3 客户端程序设计 ····· 491
22.4.4 多客户端同时访问服务器端程序 ····· 493
22.5 基于 UDP 的网络编程 ····· 494
22.5.1 UDP 通信模型 ····· 494
22.5.2 接收数据报程序设计 ····· 495
22.5.3 发送数据报程序设计 ····· 496
22.5.4 运行 UDP 网络程序 ····· 498
22.6 小结 ····· 498

第 23 章 Java 数据库应用程序开发基础 ····· 499

在本章中，介绍了 Java 的 JDBC 技术，以及如何使用 java.sql 包中的相关类编写数据库应用程序，最后还介绍了事务管理。其中应用 java.sql 包中的类编写数据库应用程序应是读者本章学习的重点。而在数据库程序中处理事务是本章学习的难点，读者应结合本书所提供的示例上机调试运行，以帮助理解相关的概念。

23.1 JDBC 技术概述 ····· 499
23.1.1 JDBC 技术介绍 ····· 499
23.1.2 JDBC 的 4 种驱动程序 ····· 500
23.1.3 如何获得 JDBC 驱动程序 ····· 501
23.1.4 设置环境变量使得驱动程序可被访问 ····· 501
23.2 java.sql 包 ····· 501
23.3 访问数据库 ····· 504
23.3.1 创建 Access ODBC 数据源 ····· 504
23.3.2 创建数据库连接 ····· 506
23.3.3 对数据库进行查询 ····· 507
23.3.4 增加、修改、删除数据库中数据 ····· 508
23.3.5 使用 PreparedStatement 访问数据库 ····· 509
23.4 事务 ····· 509
23.5 小结 ····· 511

第 24 章 使用 Swing 组件创建数据库应用程序 ····· 512

在本章中，通过若干实例学习了如何使用 Swing 组件 JComboBox、JList 和 JTable 创建数据库应用程序。读者在学习本章内容时，重点掌握 JComboBox 组件、JList 组件和 JTable 组件显示数据的方式。而这些组件所采用的 MVC 模式可能是读者学习的一个难点。建议读者将本章的示例上机调试和运行，细心体会 MVC 开发的模式。

24.1 使用 JComboBox 组件创建数据库应用程序 ····· 512
24.1.1 创建一个 JComboBox ····· 512
24.1.2 利用 DefaultComboBoxModel 创建 JComboBox ····· 515

24.2 使用 JList 组件创建数据库应用程序 …… 516
24.2.1 利用 DefaultListModel 创建 JList …… 516
24.2.2 利用 ListModel 与 AbstractListModel 创建 JList …… 518
24.3 使用 JTable 组件创建数据库应用程序 …… 519
24.3.1 与 JTable 相关的类 …… 520
24.3.2 利用 DefaultTableModel 创建 JTable …… 521
24.4 小结 …… 522

第 5 篇 Java 实战篇

第 25 章 案例 1：拼图游戏 …… 524

在本章中，通过一个完整的拼图游戏的开发过程，介绍了创建 Java 应用程序的步骤和方法。重点对游戏的算法设计和代码实现做了详细的说明。在本例中，综合运用到了 Java 的 Swing 编程、继承类、集合和泛型的使用、事件处理等知识点，让读者真正达到独立开发的能力和综合运用多种知识的能力。

25.1 软件简介 …… 524
25.1.1 应用背景 …… 524
25.1.2 功能介绍 …… 524
25.2 素材准备 …… 525
25.3 算法设计 …… 525
25.3.1 加载和拆分图片 …… 525
25.3.2 实现图片的随机排列 …… 526
25.3.3 实现图片的移动 …… 526
25.3.4 参考图像的缩放 …… 527
25.4 开发步骤 …… 528
25.4.1 自定义按钮类设计：ImageButton 类 …… 528
25.4.2 游戏界面布局：构造窗体类 PuzzleGame …… 528
25.4.3 加载和拆分图像 …… 529
25.4.4 创建拼接图面板 …… 530
25.4.5 创建显示参考图像的面板 …… 531
25.4.6 创建菜单栏 …… 533
25.4.7 实现按钮事件监听器类 …… 534
25.4.8 设置窗体在屏幕中出现的位置 …… 535
25.4.9 编写 main 方法 …… 535
25.4.10 运行测试 …… 536
25.5 打包部署 …… 536
25.6 小结 …… 536

第 26 章　案例 2：心心网络聊天室……537

在本章中，通过一个完整的网络聊天室的开发过程，介绍了创建 Java 网络应用程序的步骤和方法。重点对程序的设计和功能模块划分及代码实现做了详细的说明。在本例中，综合运用到了 Java 的 Socket 编程技术、多线程处理等知识点，其中的多线程实现双向通信是本章学习的难点，读者可参考书中提供的示例源代码，实际上机调试并运行，以帮助理解。

26.1　软件简介……537
26.1.1　应用背景……537
26.1.2　需求分析……537
26.2　设计思路与原理……539
26.2.1　服务端……539
26.2.2　客户端……540
26.2.3　建立用户界面……540
26.3　业务流程图……541
26.4　应用程序实现……541
26.4.1　建立服务器：ChatServer 类……542
26.4.2　通信处理：ChatHandler 类……543
26.4.3　建立客户端：ChatClient 类……545
26.4.4　消息封装：Message 类……548
26.5　软件部署……549
26.5.1　服务器端程序打包与部署……550
26.5.2　客户端程序打包与部署……550
26.5.3　运行测试……551
26.6　小结……551
第 27 章　案例 3：惠文教育教务管理系统……552

数据库应用程序的开发，是 Java 程序员最经常接触到的项目类型之一。数据库开发技术，同样也是 Java 语言中最重要的技术之一。本案例通过一个真实的惠文教育教务管理系统，着重向读者介绍了从项目背景、需求分析、总体设计和概要设计、数据库设计与实现、功能代码实现到应用程序部署和运行测试的完整的开发过程，比较详细地向读者展示了 Java 数据库应用程序的开发技术。

27.1　背景与主要技术介绍……552
27.1.1　应用背景……552
27.1.2　安装和配置 MySQL 数据库……552
27.1.3　配置 MySQL 数据库的 jdbc 驱动程序……557
27.2　总体设计与概要说明……558
27.2.1　功能模块划分……558
27.2.2　功能模块说明……559
27.3　业务流程图……559
27.3.1　登录模块流程……559

27.3.2 班主任管理模块流程 560
27.3.3 教务专员管理模块 560
27.3.4 人事专员管理模块 560
27.4 数据库设计 560
27.4.1 数据库需求分析 561
27.4.2 数据库概念结构设计 561
27.4.3 数据库逻辑结构设计 563
27.4.4 数据库结构的实现 565
27.5 应用程序实现 567
27.5.1 创建应用程序对象 567
27.5.2 实现登录模块 569
27.5.3 实现菜单栏 578
27.5.4 实现修改密码模块 580
27.5.5 实现“关于”对话框 587
27.5.6 实现人事管理模块 589
27.5.7 实现 DepartmentInfoBean、TeacherInfoBean 信息封装类 605
27.5.8 实现 DepartmentAction、TeacherAction 业务处理类 607
27.6 软件部署 613
27.7 项目总结 616

1

第 1 篇
Java 语言基础篇

- 第 1 章　Java 语言概述
- 第 2 章　Java 语言基本语法
- 第 3 章　数组
- 第 4 章　程序流程控制语句
- 第 5 章　数字、字符和字符串处理

第 1 章 Java 语言概述

Java 语言是由 Sun 公司开发的一种应用于分布式环境的程序设计语言，于 1995 年推出。Java 语言具有跨平台的特性，它编译后的程序能够运行在多种类型的操作系统平台上。由于它具有“一次编写，到处运行”的特点，所以特别适合于分布式环境的纯面向对象的程序设计。本章将介绍 Java 语言的背景、特点、开发环境、开发过程及开发工具的使用。



1.1 Java 语言简介

现在，到处都在谈论 Java 技术，很多软件公司也都在大量招聘 Java 软件工程师，涉及开发的范围涵盖了桌面程序、企业级应用程序和嵌入式应用程序。Java 程序设计语言成为许多从事软件开发工作的程序员的首选开发语言。下面的章节将对 Java 技术的发展历史、特点及应用进行介绍。

1.1.1 Java 语言的起源

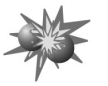
Java 语言的前身是名叫 Oak 的语言，起源于 Sun 公司的 Green 项目。1991 年，Sun 公司为了占领智能消费型电子产品的市场，资助了一个“绿色项目”，即 Green 项目。这个项目由 Sun 公司的 James Gosling 负责，最初的目的是开发嵌入式家用电器（如微波炉、遥控器、机顶盒、家电控制芯片等）的分布式软件系统，用于智能消费型电子产品，使电器更加智能化。

为此，Gosling 领导的 Sun 工程师小组决定开发一种可移植的、独立于平台的语言。使用该语言开发的程序可以运行于不同环境、不同 CPU 芯片上，即开发“一次编写，到处运行”的语言。该语言最初命名为 Oak 语言。后来 Sun 公司重新命名自己开发的这种新的语言为 Java。

到 1993 年，万维网（WWW，world wide web）和 Internet 开始蓬勃发展起来，但是 Web 应用却非常单调，只有静态网页。这时，因 Java 语言天生的平台独立性，非常适合网络编程，于是 Sun 公司将 Java 的应用转向互联网，对 Java 语言进行不断完善。

1996 年年初，Sun 公司正式推出了 Java 1.0 版。因为 Java 出色的网络功能，使得开发动态网页及为网页添加动画更加容易，为当时的互联网带来一股清新之风。

1998 年，Sun 公司又发布了全新的 Java 1.2 版，它强化了 Java 的图形处理能力，增加



了大量新的类库。因为它以功能全面、具有高度扩展能力的新版本换掉了原来的 GUI 界面和图形工具包，所以 Sun 公司将它命名为 Java 2（Java two）。

1999 年，Sun 公司宣布了以 Java 2 平台为核心的 J2EE、J2SE 和 J2ME 三大平台。时至今日，这三大平台迅速推进，满足着不断增长的市场需求。

Java 语言目前最新的版本是 Java SE 6。Java SE 6 除了命名方式上的改变和上述特性之外，还包含了许多新的特性，如新增的系统托盘类 SystemTray、调用桌面系统功能的 Desktop 类、表格的排序和过滤功能、Java 对象与 XML 之间的映射、对 Java 源文件的动态编译，以及轻量级 Http Server 等。根据 Sun 公司公布的开发计划，在 2009 年将推出 Java SE 7 版本。

现在，Java 语言已经成为开发和部署企业应用程序的首选语言，它共有三个独立的版本，用于开发不同类型的应用程序：

- Java SE：称为 Java 标准版，包含 Java 基础类库和语法。它主要用于开发图形用户界面（GUI）、复杂逻辑和高性能的桌面应用程序。
- Java EE：称为 Java 企业版，是在 Java SE 基础上，增加了编写企业级应用程序的类库。它是一个标准的多层体系结构，主要用于开发和部署分布式、基于组件、安全可靠、可伸缩和易于管理的企业级应用程序。
- Java ME：称为 Java 微型版，主要用于开发具有有限的连接、内存和用户界面能力的设备应用程序。如移动设备（手机）、PDA、能够接入电缆服务的机顶盒，以及各种终端和其他消费电子产品。

1.1.2　使用 Java 语言的优点

Java 语言是一种优秀的编程语言，它最大的优点是与平台无关。可以肯定的是，使用 Java 语言编写的程序会更加健壮，与其他语言相比，会要求更少的代码量。

对于使用 Java 语言进行软件开发的程序员来说，至少可以获得以下的优势：

- 容易入门。Java 程序设计语言是一种强大的面向对象的语言，但是 Java 语言很容易学习。
- 编写的代码量更少。从使用的类和方法的数量进行对比来看，用 Java 语言编写一个程序的代码量一般要比用 C++编写的同样程序小 4 倍。
- 编写的代码更健壮。Java 语言鼓励良好的编码实践和自动的内存碎片收集，帮助程序员避免内存泄漏。其面向对象、JavaBeans 组件体系结构，以及其丰富的、易于扩展的 API，让程序员可以重用已经存在的、测试过的代码并减少程序中的错误。
- 开发程序速度更快。Java 程序设计语言比 C++更简单，因此，当用它进行编写时，程序员的开发时间将会快两倍，并且要求的代码行更少。
- 避免平台依赖性。依赖于特定的虚拟机，Java 程序可以很容易地在不同的平台上进行移植。在 Windows、Solaris、Linux、Mac OS 及其他平台上，都可以使用相同的代码。
- 一次编写，到处运行。用 Java 程序设计语言编写的应用程序被编译为机器独立的字节码文件，这些字节码文件借助于 Java 虚拟机，能始终如一地运行在任何 Java 平台上，使得系统的移植、平台的迁移变得十分容易。
- 更容易地分发软件。用 Java Web Start 软件，用户将可以通过一个简单的鼠标单击就能启动 Java 应用程序。在启动时会自动地进行版本检查，确保用户总是使用 Java 软件的最新版本。如果是可用的更新，Java Web Start 软件将自动地更新其安装。

1.2 Java 语言与 Java 平台

通常所说的 Java 技术，既可以指 Java 程序设计语言，又可以指 Java 平台，在 Java 体系中，用“Java”一词通称。所以当程序员们在谈论“Java”时，会根据语义的环境来判断，对方指的是 Java 语言还是 Java 平台。本节将帮助读者理解 Java 语言和 Java 平台。

1.2.1 Java 程序设计语言

Java 程序设计语言是一种面向对象的高级语言。在 Java 程序设计语言中，所有的源代码首先被写成以.java 扩展名结尾的纯文本文件。然后这些源文件被 javac 编译器编译为.class 文件。一个.class 文件不包含本地处理器的代码，而是包含字节码（bytecodes）——Java 虚拟机（Java VM）的机器语言。然后 Java 启动工具在 Java 虚拟机的一个实例上运行应用程序。

如图 1.1 所示，MyProgram.java 是使用 Java 语言编写的一个纯文本形式的 Java 源文件，使用 Java 编译器将其编译为包含字节码的 MyProgram.class 二进制文件。此时的 MyProgram.class 不含机器的本地码，能部署到任意的平台上，然后由所在平台的 JVM 将其翻译为本地代码执行。



图 1.1 Java 软件开发过程

因为在许多不同的操作系统上都有可用的 Java VM，所以同样的.class 文件可以运行在微软 Windows、Solaris 操作系统（Solaris OS）、Linux 或者 Mac OS 上。一些虚拟机，如 Java HotSpot 虚拟机，在运行时还执行额外的步骤，提升应用程序的性能。通过虚拟机，同一应用程序可以运行在不同的操作平台上，如图 1.2 所示。

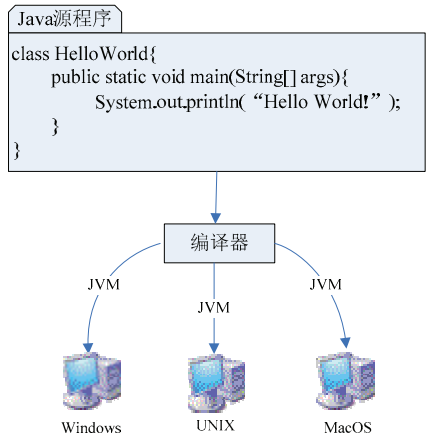


图 1.2 通过 Java VM，同一应用程序可以运行在不同的平台上

1.2.2 Java 平台

平台指的是一个程序运行所处的硬件和软件环境。前面提及了一些最流行的平台，如



微软 Windows、Solaris OS、Linux 和 Mac OS 等。大多数平台可以被描述为一个操作系统和其基础硬件的组合体。Java 平台不同于大多数的其他平台，它是一个纯软件平台，运行在其他基于硬件的平台的顶层，如图 1.3 所示，Java 平台有两个组成部分。

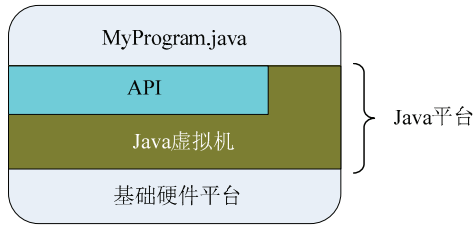


图 1.3　Java 平台

- Java 虚拟机。
- Java 应用程序设计接口（API）。

Java 虚拟机是 Java 平台的基础，被部署在各种基于硬件的平台上。而 API 是一个做好的现成软件组件的大集合，这些做好的现成软件组件提供许多有用的功能。它们被组织为相关类和接口的库，这些库就是众所周知的“包”。

作为一个平台独立的环境，一般认为，Java 平台相比本地代码有一点慢。然而，编译器和虚拟机技术的进步使得其与本地代码性能上已经非常接近。



Java 虚拟机和 JVM，都代表一个用于 Java 平台的虚拟机器。



1.3　搭建 Java 程序开发环境

在真正使用 Java 语言编写程序之前，需要先把相应的开发环境搭建好。开发环境搭建包括下载并安装 Java 开发工具包（Java SE Development Kit，JDK）、安装运行时环境及配置环境变量。安装了 JDK 以后，才能对编写的 Java 源程序进行编译，而安装了运行时环境以后才能运行二进制的.class 文件。

1.3.1　系统要求

JDK 是由 Sun 公司提供的一种用于构建 Java 应用程序、Java 小应用程序（又称为 applet）和组件的开发环境，其中包含了 Java 开发所必需的常用类库。在 JDK 中，还带有对 Java 源文件进行编译的编译器工具 javac.exe 和运行 Java 程序的 java.exe 工具，所以 JDK 对于 Java 程序开发者来说是必备的。

要在 Windows 平台下编写并运行 Java 程序，对操作系统、开发工具有如下要求。

1．操作系统要求

在 Windows 中开发 Java 应用程序，要求至少是如下的操作系统之一：Windows XP Professional、Windows XP Home、Windows 2000 Professional、Windows Server 2003、Windows Vista。

2．Java SE 开发工具箱（JDK 6）

在本书中，使用的是应用于微软 Windows 操作系统的 JDK，当前的版本是 JDK 6。在下一节将介绍如何下载并安装用于 Windows 操作系统的 JDK 6 开发工具箱。

3．一个文本编辑器

在本书的示例中，将使用 Windows 自带的记事本，作为编写 Java 程序的简单的编辑器。有 JDK 和记事本这两个工具，就足以编写第一个 Java 应用程序了。

1.3.2 下载 Java 程序开发工具包 JDK

在开发 Java 程序之前，先要在本机上安装 Java 程序开发工具包 JDK。在 Sun 公司的网站上有免费的 JDK 可供下载。读者可以打开“http://java.sun.com/javase/downloads/index.jsp”下载页面选择下载，具体步骤如下。

① 打开浏览器，在地址栏中输入网址“http://java.sun.com/javase/downloads/index.jsp”，并在键盘上按下 Enter 键，进入 JDK 下载中心页面，如图 1.4 所示。

② 在图 1.4 所示页面中，在“Java SE Development Kit（JDK）6 Update 11”这一行后面，选择【Download】选项，转到【Downloads Center】（下载中心）页面，如图 1.5 所示。

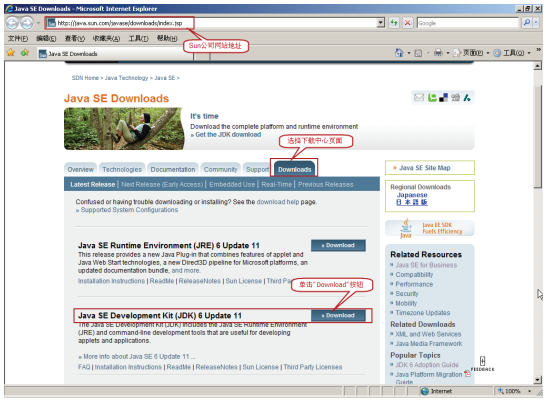


图 1.4 JDK 下载中心页面



图 1.5 下载中心页面

③ 在图 1.5 中选择要下载的 JDK 所运行的平台，这里选择【Windows】平台；同时选中版权声明所列条款前的复选框，才能继续下载，如图 1.6 所示。

④ 单击【Continue】按钮，系统会自动进入 JDK 的下载页面，如图 1.7 所示。



图 1.6 选择 JDK 所适用的平台及同意许可协议



图 1.7 选择要下载的 JDK

⑤ 在图 1.7 中所示的页面中，单击【jdk-6u11-windows-i586-p.exe】超链接，自动下载用于 Windows 平台的 JDK 安装程序。下载后的 JDK 是一个可执行文件：jdk-6u11-windows-i586-p.exe。将下载的 JDK 保存在相应的文件夹中。

说明

因为 Sun 的 JDK 版本更新很快，所以读者在实际下载时，可能具体的 JDK 名称与此稍有不同，这不影响本书程序的正确运行。



1.3.3　安装 JDK

JDK 安装包下载完成后，就可以在需要编译和运行 Java 程序的机器中安装 JDK 了。只有安装了 JDK，才能使用其中的编译工具软件对 Java 源程序进行编译，使用其中的解释执行工具来运行 Java 字节码程序。安装 JDK 的具体步骤如下。

① 关闭所有正在运行的程序，双击下载的【jdk-6u11-windows-i586-p.exe】安装文件，弹出【许可证协议】对话框，安装向导要求接受 Sun 公司的许可协议。单击【接受】按钮，接受许可协议，如图 1.8 所示。

② 选择接受许可协议后，自动打开【自定义安装】对话框，如图 1.9 所示。在这里，可以设置 JDK 的安装路径及选择要安装的功能。单击其中的【更改】按钮，可以选择 JDK 的安装路径。这里不做更改，使用默认安装路径：C:\Program Files\Java\jdk1.6.0_11。



图 1.8　接受许可协议

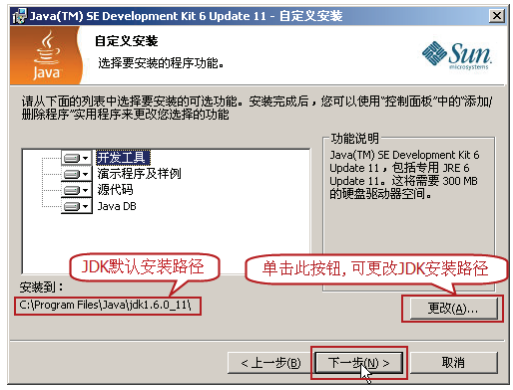


图 1.9　选择安装组件与安装路径

③ 单击图 1.9 中的【下一步】按钮，选择所需安装的 JDK 组件。此处选择默认选项。再单击【下一步】按钮，开始安装 JDK，如图 1.10 所示。

④ 在安装过程中会出现安装运行时环境 jre 的对话框，如图 1.11 所示。此处不做更改，选择默认设置。单击【下一步】按钮，继续安装。

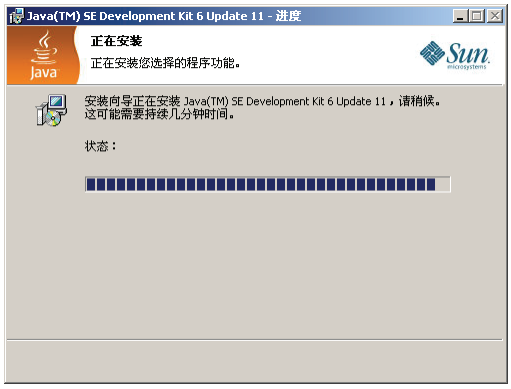


图 1.10　安装过程

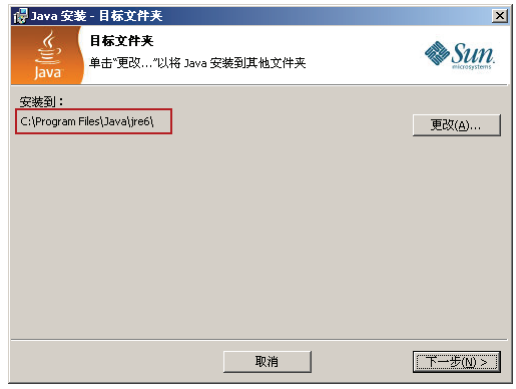


图 1.11　安装运行时环境

⑤ 安装成功以后会出现相应的提示信息。

1.3.4 在 Windows 系统下配置 JDK

读者按上一节的操作步骤完成以后，就已经成功地安装了 JDK，但是，要想正确地使用 JDK 中的类库和编译器及程序启动工具，还需要手动设置 JDK 环境变量。在 Windows 操作系统下设置 JDK 环境变量的具体操作步骤如下。

① 选择【控制面板】|【系统】|【高级】|【环境变量】命令，打开【环境变量】对话框。在【系统变量】列表框中进行环境变量的设置，如图 1.12 所示。

② 单击窗口下方的【新建】按钮，弹出如图 1.13 所示的【新建系统变量】对话框。在对话框中输入变量名“JAVA_HOME”，输入变量值为用户所安装 JDK 的路径（本书为“C:\Program Files\Java\jdk1.6.0_11）。单击【确定】按钮，保存设置，如图 1.13 所示。

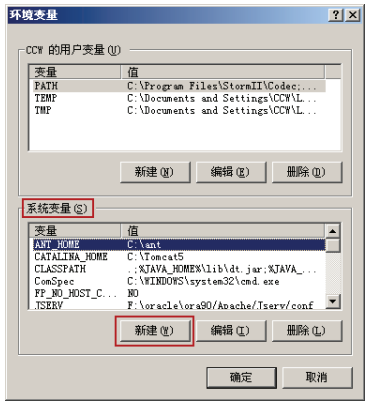


图 1.12 新建系统环境变量

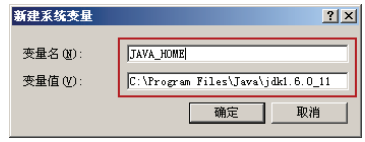


图 1.13 新建 JAVA_HOME 环境变量

③ 选中窗口下方的【Path】变量，单击【编辑】按钮，将光标移到变量值的开始位置。输入“%JAVA_HOME%\bin”，并以分号与其后面变量值隔开。单击【确定】按钮，保存设置，如图 1.14 所示。

④ 单击【新建】按钮，打开【新建系统变量】对话框。在对话框中，输入变量名“CLASSPATH”，在变量值中添加“.; %JAVA_HOME%\lib\dt.jar; %JAVA_HOME%\lib\tools.jar”。单击【确定】按钮，保存对 CLASSPATH 变量的设置。注意，输入的变量值最前面是一个点，用分号将其与后面的路径隔开，如图 1.15 所示。

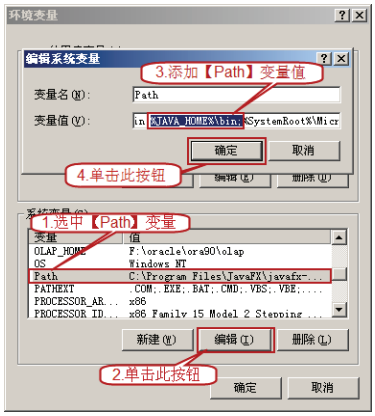


图 1.14 编辑 Path 环境变量

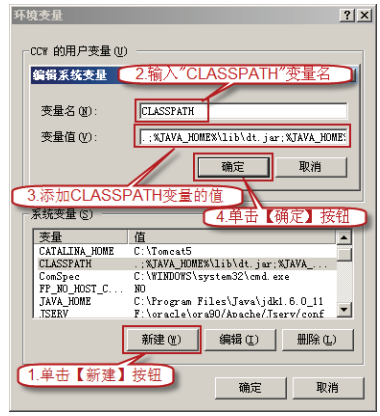


图 1.15 创建 CLASSPATH 环境变量

⑤ 单击【环境变量】对话框中的【确定】按钮，保存设置。

⑥ 测试环境变量是否配置成功。选择【开始菜单】|【所有程序】|【附件】|【命令提示符】命令，打开【命令提示符】窗口。在光标处输入命令“java -version”，并按下 Enter 键。如果出现 JDK 的版本说明，如图 1.16 所示，则说明环境变量配置成功；否则，请重新配置环境变量。



图 1.16　测试环境变量配置是否成功



重新配置环境变量以后，需要另打开一个新的【命令提示符】窗口进行测试，而不能在原来的【命令提示符】窗口下继续测试。



1.4　开发第一个 Java 应用程序

安装好 JDK 及配置好环境变量以后，就可以着手开发 Java 应用程序了。开发 Java 程序要经过以下三个步骤。

（1）创建一个源文件。

（2）将源文件编译为一个.class 文件。

（3）运行程序。

1.4.1　创建第一个 Java 应用程序源文件

Java 应用程序的开发过程，如 1.2.1 节中的图 1.1 所示。Java 源文件包含有程序代码，以 Java 语言编写。Java 源文件是纯文本文件，扩展名为“.java”。可以使用任何文本编辑器来创建和编辑源文件，本书使用 Windows 系统自带的记事本作为 Java 源文件的编辑器。

下面按上面所述步骤开发第一个简单的 Java 应用程序 HelloWorldApp.java。这个程序很简单，仅仅在控制台输出“Hello World!”。首先需要创建一个 Java 的源程序文件。

【例 1.1】创建 HelloWorldApp.java 应用程序，该程序输出“Hello World!”字符串信息。按以下步骤创建。

① 启动编辑器。通过选择【程序】|【附件】|【记事本】命令，启动记事本编辑器。在一个新的文本文档中，输入以下代码：

```
// HelloWorldApp 应用程序简单地输出一个问候："Hello World! "
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //输出"Hello World!"字符串
    }
}
```



必须准确地输入代码、命令和文件名，因为编译器 javac 和启动程序 java 都是大小写敏感的，所以必须保持大小写一致。HelloWorldApp ≠ helloworldapp。另外，文件中的所有标点符号，必须在英文状态下输入。

② 将代码保存到以 HelloWorldApp.java 名字命名的文件中。在记事本中，选择【文件】|【另存为】命令，打开【另存为】对话框。

- 使用【保存在】组合框，指定要保存文件的目录。在本示例中是“C:\examples\chapter01”目录。
- 在【文件名】文本框中输入“HelloWorldApp.java”。
- 从【保存类型】下拉列表框中，选择“文本文档（*.txt）”。
- 在【编码】下拉列表框中，保持编码为 ANSI。

结束以上操作时，对话框看上去应该如图 1.17 所示。

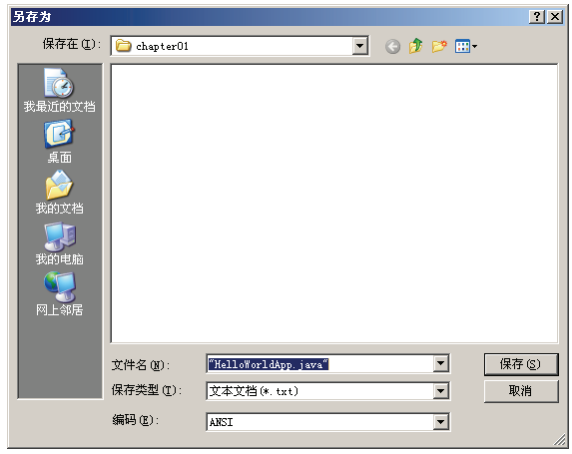


图 1.17　保存 Java 源文件

③ 打开“C:\examples\chapter01”目录，可以看到一个名为 HelloWorldApp.java 的文件。

1.4.2　将 HelloWorldApp.java 源文件编译为.class 文件

源文件是使用人类易于理解的语言和语法形式编写的，是程序人员阅读的程序代码，但计算机并不能识别，计算机只能识别“0、1”组成的二进制代码。要想让计算机识别并运行 Java 程序，还需要将源文件编译为二进制的字节码文件。

编译 Java 源文件，使用 JDK 自带的编译器工具 javac.exe，它会读取源文件并将其文本翻译为 Java 虚拟机能理解的指令，保存在以后缀.class 结尾的文件中。包含在这个文件中的指令就是众所周知的字节码（bytecodes），它是与平台无关的二进制文件，执行时由解释器 java.exe 解释成本地机器码，边解释边执行。按以下步骤编译 HelloWorldApp.java 源文件。

① 打开命令行窗口。通过选择【开始】|【所有程序】|【附件】|【命令提示符】命令，打开【命令提示符】窗口。也可以选择【开始】|【运行】命令，打开【运行】窗口。在【运行】窗口中输入“cmd”命令并回车，打开【命令提示符】窗口，如图 1.18 所示。

提示符显示了系统当前目录。系统当前目录通常是 Windows XP 的 home 目录。

② 改变当前目录。要编译源文件，改变当前目录到 HelloWorldApp.java 文件所在的目录。在本示例中，源文件是保存在“C:\examples\chapter01”目录中。输入以下命令，如图 1.19 所示。

```
cd C:\examples\chapter01
```



图 1.18　启动命令提示符窗口

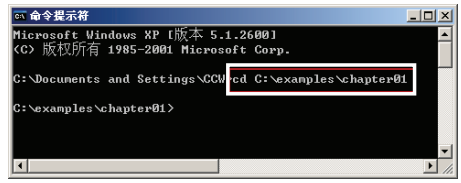


图 1.19　进入源文件所在目录

③ 查看源文件。输入“dir”命令，列出当前目录下的文件清单，如图 1.20 所示。可以看到当前目录下的源文件 HelloWorldApp.java 源文件。

④ 开始编译。在命令提示符下，输入以下的命令并回车，如图 1.21 所示。

```
javac HelloWorldApp.java
```



图 1.20　使用 dir 命令查看当前目录下文件

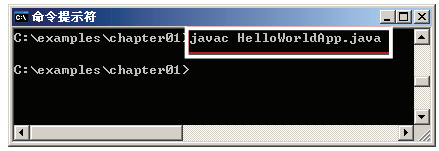


图 1.21　编译 HelloWorldApp.java 源文件

⑤ 查看.class 类文件。在命令提示符下，输入“dir”命令并回车，可以看到多了一个新的文件 HelloWorldApp.class，如图 1.22 所示。

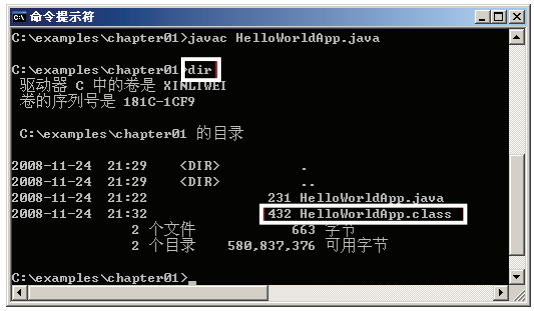


图 1.22　编译后的字节码（.class）文件

1.4.3　运行 HelloWorldApp 应用程序

将源文件编译为.class 文件后，就可以使用 Java 解释器（java.exe）来解释执行 Java 应用程序的字节码文件（.class 文件）了。Java 解释器使用 Java 虚拟机来运行 Java 应用程序。在与源文件同一目录下，在命令提示符状态下，输入以下命令并回车，应该看到如图 1.23 所示内容。

```
java HelloWorldApp
```

从图 1.23 可以看到，运行 HelloWorldApp 应用程序，输出了一个问候信息“Hello World!”。祝贺你！你的第一个 Java 应用程序工作正常！

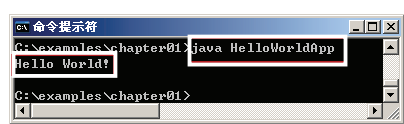


图 1.23　运行 HelloWorldApp 应用程序

1.4.4 Java 应用程序的基本结构

在成功地编写、编译并运行了第一个 Java 应用程序“HelloWorldApp.java”程序以后，下面分析一下该应用程序的基本结构。HelloWorldApp 应用程序由三个主要部分组成。

- 源代码注释；
- HelloWorldApp 类定义；
- main()方法。

1．源代码注释

Java 语言鼓励程序员在自己的程序中添加注释，用来解释程序的某些部分如何工作或某部分的特殊功能，以帮助自己或团队成员理解代码的含义。下面的程序中定义了“Hello World!”应用程序的注释。

```
// HelloWorldApp 应用程序简单地输入一个问候："Hello World! "
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //输出"Hello World! "字符串
    }
}
```

注释的内容会被编译器所忽略，也就是说，注释并不会影响程序的执行，但是对于程序员理解代码来说是很有用的。Java 语言支持三种类型的注释：单行注释、多行注释和文档注释。

- 单行注释。单行注释以“//”开始，并以换行符结束。通常对程序进行简单的说明用单行注释。编译器会忽略从“//”到本行结束间的任何内容。
- 多行注释。多行注释以“/*”开始以“*/”结尾。编译器会忽略从“/*”到“*/”这两个注释符之间的任何内容。注释的内容可以包括多行。
- 文档注释。文档注释以“/**”开始，并以“*/”结束。编译器忽略这一类的注释，就像它忽略使用“/*”和“*/”的注释一样。对于在方法体外的文档注释，可以使用 Java 的文件生成工具 javadoc 自动地将其生成文档。

2．HelloWorldApp 类定义

下面是定义 HelloWorldApp 应用程序类的代码：

```
public class HelloWorldApp {
    …            //这里为类中的属性和方法定义
}
```



HelloWorldApp 为类的名称。Java 程序是由类（class）组成的，一个 Java 源文件中可以包含多个类。在本示例中，只有一个类 HelloWorldApp 。

类的基本结构是：

```
class 类名{
  …
}
```

class 是 Java 语言的关键字，是用来定义类的，这是必需的。public 也是 Java 语言的



关键字，它将这个类定义为一个公共类（后续章节有详细和系统的讲解）。在一个 Java 程序中，可以有多个类，但是只能有一个类是公共类。在类名后面，紧跟一对花括号，花括号之间的内容称为类体。

3．main()方法

在 Java 语言中，每一个应用程序必须包含一个 main()方法，如以下代码所示。

```
// HelloWorldApp 应用程序简单地输入一个问候："Hello World! "
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //输出"Hello World! "字符串
    }
}
```

说明：

- main()方法是 Java 程序的入口，所有的 Java 应用程序都通过调用 main()方法开始执行。
- public static void main（String[] args）为固定用法，称为 main 方法的“方法签名”。其中 public、static、void 都是 Java 的关键字，其含义在后续章节中有详细讲解。
- 修饰符 public 和 static 可以以任何顺序书写（public static 或 static public），但是习惯上是使用 public static，如上面的例子中所示。
- main()方法后面，必须跟一对花括号。花括号及其之间的内容，称为“方法体”。
- 一个 Java 应用程序必须有一个类含有 main()方法，这个类称为应用程序的“主类”。
- 最后，使用核心类库中的 System 类来打印输出“Hello World!”消息到标准输出，如下面这行代码所示。

```
System.out.println("Hello World!);
```

1.5　小结

本章介绍了 Java 语言的背景、特点、开发环境、开发过程及开发工具的使用。在学习本章内容时，要重点理解 Java 技术的两个含义：Java 程序设计语言与 Java 平台。重点掌握如何搭建 Java 程序的开发环境，包括下载及安装 Java 开发工具包 JDK、配置环境变量及 JDK 环境测试等。

通过对本章的学习，读者不仅应对 Java 语言有初步的认识，还应该掌握 Java 开发环境的搭建及 Java 应用程序的开发过程。在第 2 章中，将向读者进一步介绍 Java 语言的基本语法。

第 2 章　Java 语言基本语法

学习 Java 语言和学习英语、汉语等语言一样，也要从基础语法学起，培养扎实的面向对象语言的基础。Java 语言的基本语法与 C 和 C++非常类似，但是 Java 语言是纯面向对象的编程语言，它抛弃了 C++中不容易使用的一些语言特性（如指针），因而更加简洁和易学。本章将详细介绍 Java 语言的基本语法。



2.1　标识符和关键字

一个 Java 程序是由类和对象组成的，而类和对象又是由属性（通常用变量代表）和方法（通常用函数代表）等组成的。在编写 Java 应用程序时，需要大量地为程序、类、对象、方法和变量等起名字，这就需要了解标识符和关键字的概念，以及标识符的命名规则。

2.1.1　标识符

Java 语言中，用来标识类名、对象名、变量名、方法名、类型名、数组名、文件名的有效字符序列，称为“标识符”。简单地说，标识符就是一个名字。Java 语言规定，标识符由字母、数字、下画线和美元符号组成，并且第一个字符不能是数字。例如，下列标识符都是合法的。

```
Dog   class_3   _price  $123  $23boy
```

下列标识符为非法的。

```
3_class   No.1   ￥123
```



Java 语言是区分大小写的，所以标识符 boy 与 Boy 是不同的。

2.1.2　关键字

在 Java 语言中，还有一些专门的词汇，Java 语言已经赋予了特定的含义，只能用于特定的地方。所以程序员在编写程序时，不能再使用这些词汇来命名标识符，不能改变这些词汇的含义。这些专有词汇，称为“关键字”。

例如在社会生活中，“派出所”这个词已经被赋予了特定的含义，所以如果有人开个商



店，他绝对不能给商店取名为“派出所”。在这里，“派出所”就是社会生活中的关键字。在 Java 语言中同样如此。对于有特定含义的关键字，在编程时不能用来命名自定义的标识符。

Java 的关键字对 Java 的编译器有特殊的意义，是 Java 语言中为语法定义的需要而特别定义的标识符。这些标识符构成了 Java 语言最基本的语素，它们用来表示一种数据类型，或者表示程序的结构等。如表 2.1 中列出了 Java 语言的所有关键字。

表 2.1　Java 语言关键字列表

abstract	boolean	break	byte	case	catch	char	class
const	continue	default	do	double	else	enum	extends
final	finally	float	for	goto	if	implements	import
instanceof	int	interface	long	native	new	package	private
protected	public	return	short	static	strictfp	super	switch
synchronized	this	throw	throws	transient	try	try	void
volatile	while						



true、false、null 不是关键字，是保留字，但是仍然不能用于命名标识符。保留字是为 Java 预留的关键字，它们虽然现在没有作为关键字，但在以后的升级版本中有可能作为关键字。

2.1.3　标识符命名规则

为了提高程序的可读性，在程序中定义标识符时，要尽量遵循“望文生义”的原则，即名称要能代表变量本身的含义。例如定义一个标识符代表价格，那么取名为“price”，就会让其他人看到这个标识符就能明白它代表价格。Java 标识符的具体命名规则如下。

- 一个标识符可以由几个单词连接而成，以表明它所代表的含义，如 applePrice。
- 如果是类名，每个单词的首字母都要大写，其他字母则小写，如 StudentInfo。
- 如果是方法名或者变量名，第一个单词的首字母小写，其他单词的首字母都要大写，如 getStudentInfo()、studentName。
- 如果是常量，所有单词的所有字母全部大写，如果由多个单词组成，通常情况下单词之间用下画线“_”分隔，如 PI、MAX_VALUE。
- 如果是包名，所有单词的所有字母全部小写，如 examples.chapter02。

2.2　常量与变量

在编写程序时，经常接触到不同类型的数据。有的数据在程序运行期间是不允许改变的，如数学运算中的圆周率π；有的数据在程序运行期间是需要经常改变的，如价格、数量等。这就需要在程序中有相应的表示。

2.2.1　常量的概念及常量声明

常量是指在程序执行期间其值不能发生变化的数据。常量是固定的，其值在程序运行期间是不能改变的。如整数 123，小数 1.23，字符'A'，布尔常量 true、false 等。在 Java 中，常量分为字面常量和字符常量两种。

- 字面常量：指的是如 123、12.34、‘m’、true 这样字面上本身就是一个固定值的数据。

- 字符常量：指的是代表一个固定值的标识符。使用字符常量需要在程序中先声明后使用。

所谓声明一个常量，指的是创建一个常量，为它起一个名字，并指明它所代表的数据类型。如果要声明一个字符常量，就必须使用关键字 final。声明字符常量的语法如下。

```
final 常量类型  常量标识符=常量值;
```

在声明常量时，通常情况下立即为其赋值，即立即对常量进行初始化。例如：

```
final float PI = 3.14F;              //声明一个 float 类型常量，并初始化为 3.14
final int STUENT_NUMBER = 24;        //声明一个整型常量，并初始化为 24
```

在声明常量标识符时，按照 Java 的命名规则，所有的字符都要大写。如果常量标识符由多个单词组成，则在各个单词之间用下画线“_”分隔，如上例中的 STUENT_NUMBER。

在上面所示的例子中，是在声明符号常量时，进行初始化的。也可以先声明常量，再进行初始化，例如：

```
final float PI;                      //声明一个 float 类型常量
final int STUENT_NUMBER;             //声明一个整型常量
PI = 3.14F;                          //初始化为 3.14
STUENT_NUMBER = 24;                  //初始化为 24
```

但是初始化以后，就不允许再在程序中对这些常量进行重新赋值，即不允许改变常量的值。例如：

```
final int STUENT_NUMBER = 24;   //声明一个整型常量，并初始化为 24
STUENT_NUMBER = 32;             //企图修改常量 STUENT_NUMBER 的值，程序将报错
```

如果需要声明多个同一类型的常量，可以使用下面的语法：

```
final 常量类型  常量标识符 1,常量标识符 2,常量标识符 3，…;
final 常量类型  常量标识符 1=常量值 1,常量标识符 2=常量值 2,常量标识符 3=常量值 3;
```

例如：

```
final float PI，PRICE，WEIGHT;       //声明三个 float 类型的变量
final float PI=3.14F,price=13.86F,WEIGHT=86.32F;
                                     //声明三个 float 类型的变量，同时进行初始化
```

2.2.2 变量的概念及变量声明

所谓变量，就是值在程序运行期间可以被改变的量。实际上，变量是程序在运行时内存中能够存储信息的地方，它的值可以在程序的任何地方被动态地改变。所以在编写程序时，经常需要声明变量，向系统申请存储数据的内存空间。

在程序中使用变量，必须先创建它并为它取一个名字，并且指明它能够存储信息的类型，这称为“变量声明”。要在程序中声明一个变量，其语法形式如下。

```
变量类型  变量标识符;
```

变量类型会在下一节详细讲述。变量标识符遵循标识符命名规则，第一个单词的首字母小写，其他单词的首字母大写，其他字母一律小写。例如：

```
String studentName;              //声明一个 String 类型的变量，用来存放学生的姓名
int studentAge;                  //声明一个 int 型变量，用来存放学生的年龄
```

在声明变量时，可以立即为其赋值，即立即对变量进行初始化，例如：

```
String studentName = "张小明";   //声明一个 String 类型的变量，并初始化
int studentAge = 18;             //声明一个 int 型变量，并初始化
```

也可以先声明变量，然后在需要的时候再初始化，例如：

```
String studentName;              //声明一个 String 类型的变量
int studentAge;                  //声明一个 int 型变量
studentName="张小明";            //初始化赋值
studentAge=18;                   //初始化赋值
```

也可以同时声明多个同一类型的变量，如：

```
String studentName , teacherName;                    //声明两个 String 类型的变量
String studentName="张小明" , teacherName="王金柱";//声明两个 String 型变量，并初始化
```

变量与常量不同。变量的值如果需要的话，可以在程序的任何地方被改变。例如，下面的代码是正确的。

```
String studentName = "张小明";   //声明一个 String 类型的变量，并赋初值"张小明"
studentName = "李芳";            //改变变量 studentName 的值为"李芳"
```



2.3　基本数据类型

变量是程序在运行时内存中能够存储信息的地方。在现实生活中，有各种各样的数据类型，例如价格为 19.8 元（小数）、员工人数 25 人（整数）等。所以，在声明变量时，必须指明它能够存储信息的类型。不同的数据类型，在内存中分配的空间大小也不一样，也就是说，要指明变量的数据类型。基本的数据类型是构造语言的最基础的要素。本节将重点介绍 Java 语言中的各种基本数据类型。

2.3.1　数据类型

作为程序，处理的对象就是各种类型的数据。因此，程序设计语言中数据类型的多少，是衡量语言功能是否强大的标准之一。Java 语言有着非常丰富的数据类型，可分为原始数据类型和构造数据类型两大类，如图 2.1 所示。

在 Java 语言中，一共有 8 个原始数据类型，可分为以下 4 种类型，分别为：

- 整数类型；
- 浮点类型；
- 字符类型；
- 逻辑类型。

下面各节将分别对这几种原始（基本）数据类型进行详细的讲解和说明。

2.3.2　整数类型

整数类型的变量或常量，只能用来存放整数。例如，在某个程序中，要定义一个变量来保存员工的人数，毫无疑问，人数必须是整数，不可能出现 0.6 个人，所以定义这样的变量时，要将其声明为整数类型。在 Java 中，有 4 种整数类型，如表 2.2 所示。

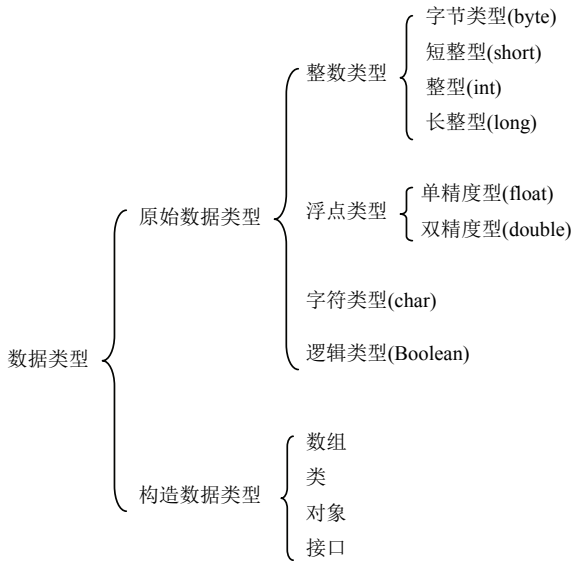


图 2.1　Java 中的数据类型分类

表 2.2　Java 中的整数类型

数据类型	关键字	分配内存大小	取值范围
字节型	byte	1 个字节	-128～127(-2^7 ～ 2^7-1)
短整型	short	2 个字节	-32768～32767(-2^{15} ～ $2^{15}-1$)
整型	int	4 个字节	-2 147 483 648～2 147 483 647(-2^{31} ～ $2^{31}-1$)
长整型	long	8 个字节	-9 223 372 036 854 775 808～-9 223 372 036 854 775 807 (-2^{63} ～ $2^{63}-1$)

所有这些类型都是有符号的，也就是说，它们既能存储正数，也能存储负数。不同类型的整型变量，内存分配给它们的空间大小也不一样。因此对于不同类型变量，它所能保存的数值的大小也是有限制的，不能超出它的取值范围。正如在上一节中讲到的，程序员在程序中声明一个变量的时候，必须同时指明变量的数据类型，用于确定这个变量可以保存值的范围。例如：

```
public class Test1{
    public static void main(String[] args){
        byte studentAge = 18;       //声明一个字节型变量，并赋予初值 18
        //studentAge=180;           //错误，超出了 byte 型变量的取值范围
        System.out.println("studentAge="+studentAge);
    }
}
```

因为变量 studentAge 声明为 byte 型的，它在内存空间中只被分配 1 个字节大小，这 1 个字节所能容纳的数值的大小为-128 ～ 127。所以为变量 studentAge 赋值 18 是可以的，而为它赋值 180 已经超出了它所能容纳数值的范围，编译时就会提示错误信息。

在为 long 型常量或变量赋值时，需要在所赋值的后面加上一个字母“L”(或小写“l”)，说明所赋的值为 long 型。如果所赋的值未超出 int 型的取值范围，也可以省略字母“L”(或小写 “l”)。例如，下面几种赋值方式都是正确的。

```
long account = 3124567893L; //所赋的值超出了 int 型的取值范围，必须在后面加上字母“L”
```

```
long account = 31245678L;   //所赋的值未超出 int 型的取值范围,可以在后面加上字母“L”
long account = 31245678;    //所赋的值未超出 int 型的取值范围，可以省略字母“L”
```

但是下面的这种赋值方式是错误的。

```
long account = 3124567893; //所赋的值超出了 int 型的取值范围，但是没有在后面加上字母“L”
```

2.3.3　浮点类型

另一种能够存储的数字类型是浮点类型。声明为浮点类型的变量或常量，主要用来存储小数。浮点类型包括两种：float 型（单精度浮点型）和 double 型（双精度浮点型），如表 2.3 所示。

表 2.3　Java 中的浮点类型

数据类型	关 键 字	分配内存大小	取值范围
单精度型	float	4 个字节	1.4E-45 ～ 3.4E+38
双精度型	double	8 个字节	4.9E-324 ～ 1.797E+308

如果要声明一个用来保存小数的变量，必须将其声明为浮点类型。例如：

```
float price = 326.85F;              //声明一个 float 类型的变量，并赋值
double height = 3836.8;             //声明一个 double 类型的变量，并赋值
double height = 3836.8D;
```

如上面的示例所示，在为 float 类型的变量赋值时，需要在所赋值的后面加上字母“F”或小写“f”，否则编译时，会提示错误。因为不加上字母“L”或小写“l”时，默认的数值为 double 类型，把 double 类型的数值赋给 float 类型的变量，是不正确的。

在为 double 类型的变量赋值时，既可以在所赋值的后面加上字母“D”或小写“d”，也可以不加。浮点型变量除了可以接受小数之外，也可以接受整数，例如：

```
float price = 326;                  //声明一个 float 类型的变量，并赋予整数值
double height = 3836;               //声明一个 double 类型的变量，并赋予整数值
```

如果所赋整数值的大小超出了 int 型的取值范围，那么也需要在所赋值后面加上相应的字符“F（f)”或“D（d)”，将其转换为小数，再赋值，例如：

```
float price = 3124567893F;          //声明一个 float 类型的变量，并赋值
double height = 3836567893D;        //声明一个 double 类型的变量，并赋值
```

将整数赋值给 float 或 double 类型的变量时，会进行数据类型的转换，这在后面的 2.3.6 节“不同数据类型间的转换”中会有详细的说明。

2.3.4　字符类型

在程序中，还经常需要保存字母、数字符号、标点符号和其他符号等，如'a'、'#'、'3'，这就需要用到另外一种数据类型——字符型。字符型变量或常量在内存中被分配两个字节的空间，用来存储单个字符。在 Java 语言中，使用关键字 char 来声明字符型常量或变量，例如：

```
char gender = '男';          //声明一个 char 类型的变量，用来存储性别信息
char star = '*';             //声明一个 char 类型的变量，用来存储符号'*'
final char PI = 'π';         //声明一个 char 类型的常量，用来存储圆周率符号'π'
```

因为计算机只能识别二进制数据，所以在 Java 中，将不同的字符映射到不同的整数数字，再转换成二进制的形式，以利于计算机对字符的识别。这种映射关系，就是“字符编码”。

在 Java 中，对字符采用 Unicode 编码。Unicode 使用 2 个字节表示 1 个字符，并且 Unicode 字符集中的前 128 个字符与 ASCII 码字符集兼容。如字符'a'，它的 ASCII 码的二进制数据表示形式为 01100001，而它的 Unicode 码的二进制数据表示形式为 00000000 01100001，它们都表示十进制数的 97。

Unicode 编码采用的是无符号编码，在内存中占 2 个字节共 16 位，一共可以存储 65536 个字符。所以在 Java 语言中，几乎可以处理世界上所有国家的语言文字，这也是 Java 特别适合互联网程序开发的一个特点。

当声明一个 char 类型的变量并为其赋值时，所赋的值必须为一个英文字母、一个符号或一个汉字，并且要用英文状态下的单引号括起来，如'男'、'*'、'π'等。

有些字符，不能通过键盘输入到程序中，如回车符、换行符等，这就需要使用 Java 中的转义字符常量。这些转义字符常量用特定的字符，表示无法在键盘上输入的字符，如表 2.4 所示。

表 2.4　Java 中的转义字符

转义字符	含　义
\n	表示换行
\t	表示横向跳格，作用同 Tab 键
\b	表示退格
\r	表示回车
\f	表示走纸换页
\\	表示反斜杠字符
\’	表示单引号字符
\”	表示双引号字符
\d	表示八进制字符
\xd	表示十六进制字符
\ud	表示 Unicode 字符

需要注意的是，转义字符虽然是反斜杠“\”和一个字符的组合，但在 Java 中，它代表一个字符，如下面的代码所示。

```
public class Test2{
    public static void main(String[] args){
        //声明一个 char 类型的变量，用来存储转义字符'\n'，代表换行
        final char changeLine = '\n';
        //输出连接后的字符串
        System.out.println("我"+changeLine+"爱"+changeLine+"你");
    }
}
```

在上面的代码中，定义一个 char 类型的变量 changeLine，并赋予一个转义字符值'\n'，然后输出一行信息。输出结果如下。可以看出，转义字符发挥了作用，每输出一个字就进行了换行。

```
我
爱
```



你

2.3.5　逻辑类型

在 Java 程序中，要保存逻辑值 true（真）或 false（假），需要声明逻辑型的常量或变量。逻辑类型又称布尔类型。声明逻辑类型常量或变量，需要使用关键字 boolean。如下面的示例所示。

```
boolean flag = false;              //声明一个 boolean 类型的变量，初始值为 false
flag = true;                       //改变 flag 变量的值为 true
```

布尔类型的变量或常量，只能接受 true 和 false 这两个逻辑值，不能向布尔类型的变量赋予除 true 和 false 之外的其他值。

2.3.6　不同数据类型间的转换

当把一种数据类型变量的值赋予另一种数据类型变量时，或者不同类型的数据或数据变量混合在一起进行运算时，就需要对数据类型进行相应的转换。这种情况在编写程序时经常遇到，例如下面这行代码：

```
System.out.println(3.6+4+'a');       //输出表达式 3.6+4+'a'的值
```

输出表达式 3.6+4+'a'的值，那么输出结果是什么呢？这就需要掌握不同数据类型之间进行转换的规则。在 Java 中，不同数据类型之间的转换，分为两种情况：自动类型转换和强制类型转换。

1．自动类型转换

在对变量进行赋值时，如果是将低级数据类型（相对取值范围较小的数据类型）的数据，赋给高级数据类型（相对取值范围较大的数据类型）的变量，这时 Java 会自动完成从低级类型向高级类型的转换。在原始数据类型中，除了 boolean 类型外，其他数据均可参与算术运算，数据类型从低到高的排列顺序如图 2.2 所示。

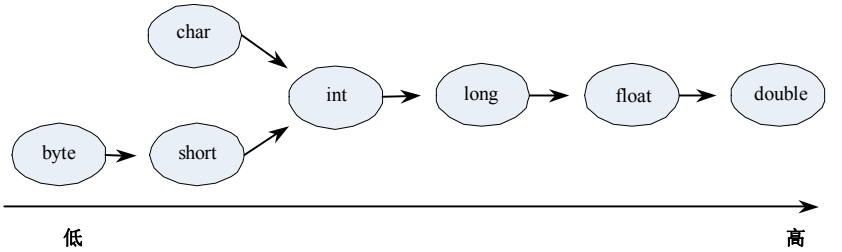


图 2.2　数据类型级别与取值范围

图 2.2 中列出了数据类型的大小关系，一个类型是否可以自动隐式转换为另一种类型，要看是否能通过箭头到达。例如下面的代码，在进行赋值运算时：

```
float price = 36;                          //定义 float 类型的变量，并赋值
System.out.println("price="+price);        //输出 price 变量的值：price=36.0
```

可以看出，因为相对于 float 类型来说，整数 36 为 int 型，属于低级数据类型，所以 Java 自动将其转换为 float 类型的 36.0，并赋值给 float 类型的变量 price。可以得出这样的

结论：将低类型的数据赋给高类型的变量时，Java 会自动将低类型的数据转换为高类型的数据，然后赋值。

在不同类型的数据进行混合运算时，分为两种情况：

➢ 参与混合运算的，只有 byte、short 或 char 类型的数据。

这种情况下，Java 首先将 byte 或 short 或 char 类型转换为 int 类型，然后再参与运算，并且运算结果也是 int 型的。例如：

```
public class Test3{
    public static void main(String[] args){
        byte     byteVar = 68;              //声明一个 byte 类型的字节变量
        short  shortVar = 873;              //声明一个 short 类型的整型变量
        char  charVar = ‘a’;                //声明一个 char 类型的字符变量
        int  value = byteVar + shortVar + charVar;
        System.out.println(“value = ” + value); //输出 value 的值:value = 1038
    }
}
```

在上面的示例中，分别将变量 byteVar、shortVar、charVar 转换为 int 型，然后相加，相加结果为 int 型，并赋值给 int 变量 value。

➢ 参与混合运算的，含有 int、long、float 或 double 型的数据。

这种情况下，Java 首先将所有低数据类型转换为表达式中数据类型最高的数据类型，然后再参与运算，并且运算结果也是表达式中最高的数据类型。例如：

```
public class Test4{
    public static void main(String[] args){
        byte  byteVar = 68;                 //声明一个字节类型的变量
        int    intVar = 873;                //声明一个 int 类型的变量
        float  floatVar = 128.5F;           //声明一个 float 类型的变量
        double  value = byteVar + intVar + floatVar;    //不同数据类型变量进
行混合运算
        System.out.println(“value = ” + value);       //输出 value 的值:
value = 1069.5
    }
}
```

在上面的示例中对 value 值的计算，最高数据类型为 double（变量 value 的数据类型），所以三个变量 byteVar、intVar、floatVar 在进行加法运算时，首先都被转换为 double 型，然后相加，相加结果也为 double 型，并赋值给 double 型变量 value。

2．强制类型转换

在对变量进行赋值时，如果是将高级数据类型的数据赋给低级数据类型的变量，这时 Java 不会自动完成类型的转换，而必须由程序员手工强制进行类型转换。例如，在进行赋值运算时：

```
int count = 36.5;                   //定义 int 类型的变量，并赋值 double 类型的 36.5
System.out.println("count="+count);
```

上述代码在编译时会提示错误，因为将高级类型的数据 36.5 赋给了低级类型的变量 count。如果改为下面这种形式：

```
int count = (int)36.5;                          //强制类型转换
```

```
System.out.println("count="+count);          //输出 count 变量的值：count=36
```

在上面的代码中，（int）显示为强制类型转换的语法，即在欲转换的数据或变量前面，用“（目的数据类型）”的形式，强制转换为括号中的目的数据类型。另外，在上面的示例中也可以看到，在进行强制类型转换时，进行了简单的截断而不是四舍五入（输出的是 36，而不是 37）。

同样，在不同类型的变量进行混合运算时，程序员也可以通过强制类型转换，让参与运算的变量改变数据类型。例如：

```
public class Test5{
    public static void main(String[] args){
        byte  byteVar = 68;                  //声明一个字节类型的变量
        int    intVar = 873;                 //声明一个 int 类型的变量
        float  floatVar = 128.5F;            //声明一个 float 类型的变量
        int  value = byteVar + intVar +(int) floatVar;      //混合运算，对
floatVar 进行强制类型转换
        System.out.println(“value = ” + value); //输出 value 的值:value = 1069
    }
}
```

在上面的程序中，将 byte 类型的变量 byteVar、int 类型的变量 intVar 和 float 类型的变量 floatVar 进行混合加法运算，并将结果赋给一个整型变量 value。其中 byte 类型低于 int 类型，所以变量 byteVar 会自动转换为 int 类型。而 float 类型高于 int 类型，所以需要强制类型转换，将其转换为 int 型。然后三个 int 类型的值相加，将结果赋给 int 类型的变量 value。

2.4　运算符与表达式

运算符指的是一些特殊的符号，它们用于数学函数、赋值语句和逻辑比较等方面。而使用运算符将运算数据连接起来的符合 Java 语法规则的式子，称为表达式。在 Java 程序中，会用到大量的运算符和表达式。Java 中，共有以下几种运算符：

- 赋值运算符与赋值表达式；
- 算术运算符与算术表达式；
- 自增和自减运算符；
- 关系运算符与关系表达式；
- 逻辑运算符与逻辑表达式；
- 位运算符；
- 三元运算符。

在下面的几节中，将会对运算符和表达式的使用进行详细介绍。

2.4.1　算术运算符与算术表达式

算术运算是人们在日常生活中经常要遇到的，相应地，在程序设计语言中也有相应表示算术运算的语法。在 Java 语言里，共有 5 种算术运算符来完成基本的算术运算，如加法运算、减法运算、乘除运算等。这 5 种算术运算符如表 2.5 所示。

表 2.5　算术运算符

运 算 符	表 达 式	含　义
+	8+3	加
-	8-3	减
*	8*3.5	乘
/	8/3	除
%	8%3	求余

算术运算符都是二元运算符，运算符两侧要求有两个操作数。用算术运算符和两边的数字或变量连接起来的式子，就是算术表达式，如：8+3、8*3.5 等。和代数中的加、减、乘、除、求余类似。这里要特别注意除法运算和求余运算。

1．除法运算符/

如果进行除法运算的两个操作数都是整数，那么不论能否整除，运算结果都将是一个整数，例如：

```
System.out.println(13/2);          //输出的运算结果为 6
System.out.println(8/3);           //输出的运算结果为 2
```

这里要注意的是，运算结果只是简单的截断，即去掉小数部分，而不是四舍五入。

2．求余运算符%

当在整数之间进行求余运算时，运算结果为数学运算中的余数。例如：

```
System.out.println(13%2);          //输出的运算结果为 1
System.out.println(8%3);           //输出的运算结果为 2
System.out.println(8%13);          //输出的运算结果为 13
System.out.println(8%2);           //输出的运算结果为 0
```

3．关于 0 的问题

与数学运算一样，0 可以做被除数，但是不可以做除数。当 0 做被除数时，无论是除法运算还是求余运算，运算结果都为 0。例如：

```
System.out.println(0/3);           //输出的运算结果为 0
System.out.println(0%3);           //输出的运算结果为 0
```

如果用 0 做除数，虽然可以通过编译，但是在程序运行时，会抛出异常 java.lang.ArithmeticException，这在有关异常处理的章节中，会有详细讲解。

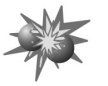
2.4.2　赋值运算符与赋值表达式

赋值运算符为“=”，即等号。赋值运算符是二元运算符，它的左边必须是变量，不能是常量或表达式。赋值运算符的含义，是将运算符“=”右边的值赋给其左边的变量。

给一个变量进行赋值的式子，就是赋值表达式，因为它产生了一个值，而整个赋值表达式的值就是等号（=）左边的值。如 x=20 和 y=true，都是赋值表达式。可以用如下的方式将赋值语句连在一起：

```
x = y = z = 13;
```

赋值运算符的运算（结合性）是自右向左的。所以，在这个语句中，三个变量 x、y、



z 的值都是 13。在进行赋值时，总是先计算赋值运算符右边的值，然后将值赋给左边的变量，所以可以像下面这样使用表达式语句：

```
int count = 6;
count = count + 5;        //首先获取 count 变量的值,与 5 相加,再将和赋值给变量 count
```

除了赋值运算符之外，Java 还提供了几个复合的赋值运算符，以提高程序员的编码效率，如表 2.6 所示。

表 2.6　赋值运算符

运 算 符	表 达 式	含　义
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

请看下面的示例：

```
public class Test6{
    public static void main(String[] args){
        int x=8,y=3;
        System.out.println("x+=y:x=" + (x+=y));     //此语句执行完以后，x=11
        System.out.println("x-=y:x=" + (x-=y));     //此语句执行完以后，x=8
        System.out.println("x*=y:x=" + (x*=y));     //此语句执行完以后，x=24
        System.out.println("x/=y:x=" + (x/=y));     //此语句执行完以后，x=8
        System.out.println("x%=y:x=" + (x%=y));     //此语句执行完以后，x=2
    }
}
```

编译并运行此程序，输出结果如下。

```
x+=y:x=11
x-=y:x=8
x*=y:x=24
x/=y:x=8
x%=y:x=2
```

要特别注意其中的除号“/”，如果该运算符两边为整数，结果也为整数。

2.4.3　自增和自减运算符

在编程中，有一种常见的任务是从一个整数变量上增加或者减小 1。Java 提供了一类特殊的运算符，用于这种表达式，称做“自增运算符（++）”和“自减运算符（--）”。使用自增和自减运算符，可以减少一定的代码量，使程序更加简洁。

++、--运算符是一元运算符，可以放在操作数之后，也可以放在操作数之前。表达式 x++或++x 相当于 x=x+1，而表达式 x--或--x 相当于 x=x-1。例如：

```
public class Test7{
    public static void main(String[] args){
        int x=8,y=3;
        System.out.println(x++);    //输出结果为 8；此语句执行完以后，x 的值为 9
        System.out.println(x);      //输出 9
```

```
        System.out.println(++y);     //输出结果为 4；此语句执行完以后，x 的值为 4
        System.out.println(y);       //输出 4
    }
}
```

从上例可以看出，当++运算符放在变量 x 后面时，先输出变量 x 的值，然后变量 x 自增 1，即由 8 自增为 9；而当++运算符放在变量 y 前面时，先将变量 y 的值增加 1，然后输出变量 y 的值。--运算符使用方式同++运算符一致。

如果++运算符放在变量名前面，那么它们被称为“前缀运算符”；如果放在变量名后面，它们就被称为“后缀运算符”。对于前缀运算符来说，它总是先自增 1，然后参与运算；对于后缀运算符来说，它总是先以原来的值参与运算，然后再自增 1。

2.4.4 关系运算符和关系表达式

在程序中，有时需要对两个值或变量进行比较，如比较两个人年龄的大小、比较两种产品销售额大小等。在 Java 语言中，使用关系运算符来完成这样的比较需要。Java 中的关系运算符如表 2.7 所示。

表 2.7 关系运算符

运 算 符	含 义	表 达 式	运算结果
>	大于	78>23	true
<	小于	‘f’<’a’	false
>=	大于或等于	5.6>=3.2	true
<=	小于或等于	‘A’>=65	true
==	等于	‘A’==65	true
!=	不等于	‘A’!=65	false

算术表达式的运算结果为 boolean 型。当关系表达式成立时，运算结果为 true；当关系表达式不成立时，运算结果为 false，如表 2.6 所示。要注意的是，当进行相等比较时，运算符为两个连着的等号“＝＝”，不要写成赋值运算符“＝”。

所有关系运算符都可以用于对整数、浮点数和字符型数据的比较，其中等于和不等于运算符可用于所有数据类型。

2.4.5 逻辑运算符和逻辑表达式

在实际编写程序时，有时需要将返回类型为 boolean（布尔）类型的表达式组合在一起，构成一个更加复杂的表达式。例如，在某程序中，需要找出库存量小于等于 10 个并且价格大于 13.5 元的商品。

```
stocks 小于等于 10 并且 price 大于 13.5
```

“stocks 小于等于 10”和“price 大于 13.5”这两个关系表达式都返回 boolean 类型的值，要将这两个表达式组合在一起，需要通过逻辑运算符来实现。逻辑运算符被用做逻辑上的组合，包括：与（&和&&）、或（|和||）、异或（^）、非（!）。

1．与运算符&和&&

当两个关系表达式通过&或者&&连接在一起时，只有当两个关系表达式的值都为真



时，该组合的表达式的值才为真（true）。看下面的例子：

```
boolean haveStocks = (stocks<=10) & (price>13.5);
System.out.println(haveStocks);
```

这个表达式将两个比较表达式“stocks<=10”和“price>13.5”连接在一起。如果表示库存量的变量 stocks 的值小于等于 10，并且表示商品价格的变量 price 的值大于 13.5 元，即两个表达式的值都为真（true），那么整个表达式“（stocks<=10） & （price>13.5）”的值才为真，并且将真值 true 赋予 boolean 类型的变量 haveStocks。

虽然&和&&都是用于“与”的运算，但是它们还是有些不同。如果使用“&&”，当“&&”左侧的表达式结果为 false 时，就不再计算其右侧的表达式，而是直接返回运算结果 false。只有当“&&”左侧的表达式结果为 true 时，才计算其右侧的表达式。请看下面的例子。

```
public class Test8{
    public static void main(String[] args){
        int a=3,b=4;            //声明两个变量 a 和 b，并初始化为 3 和 4
        boolean flag;
        flag=(a>2)&&(b++>2);    //“&&”运算符左侧运算结果为 true，右侧表达式被运算
        System.out.println("flag="+flag);  //输出结果：flag=true
        System.out.println("b="+b);        //输出结果：b=5
        b=4;                                //重置变量 b 的值
        flag=(a>5)&&(b++>2);                //“&&”运算符左侧运算结果为 false，
右侧表达式不被运算
        System.out.println("flag="+flag);  //输出结果：flag=false
        System.out.println("b="+b);        //输出结果：b=4
    }
}
```

所以，逻辑运算符&&又被称为“简洁与”或“短路与”。而运算符“&”与“&&”不同，不管“&”运算符左侧的运算结果是什么，都计算其右侧的表达式，再返回整个逻辑表达式的值。请看下面的例子。

```
public class Test9{
    public static void main(String[] args){
        int a=3,b=4;            //声明两个变量 a 和 b，并初始化为 3 和 4
        boolean flag;
        flag=(a>2)&(b++>2);     //“&”运算符左侧运算结果为 true，右侧表达式被运算
        System.out.println("flag="+flag);      //输出结果：flag=true
        System.out.println("b="+b);            //输出结果：b=5
        b=4;                                    //重置变量 b 的值
        flag=(a>5)&(b++>2);     //“&”运算符左侧运算结果为 false，右侧表达式仍
然被运算
        System.out.println("flag="+flag);      //输出结果：flag=false
        System.out.println("b="+b);            //输出结果：b=5
    }
}
```

从上例中可以看到，不论“&”运算符左侧的运算结果是什么，其右侧的表达式都会被计算。

2．或运算符“|”和“||”

当两个关系表达式通过“|”或者“||”连接在一起时，只要两个关系表达式的值有一

个为真，则该组合的表达式的值就为真（true）。只有当运算符两侧的表达式都为 false 时，整个逻辑表达式的值才为 false。看下面的例子。

```
boolean haveStocks = (stocks<=10) | (price>13.5);
System.out.println(haveStocks);
```

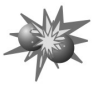
这个表达式将两个比较表达式 stocks<=10 和 price>13.5 连接在一起。如果表示库存量的变量 stocks 的值小于等于 10，或者表示商品价格的变量 price 的值大于 13.5 元，即两个表达式的值有一个为真（true），那么整个表达式“(stocks<=10) | (price>13.5)”的值就为真，并且将真值 true 赋予 boolean 类型的变量 haveStocks。

虽然“ |” 和“ ||” 都是用于“或”的运算，但是它们还是有些不同。与“&&”运算符类似，使用“||”，当“||”运算符左侧的表达式结果为 true 时，就不再计算其右侧的表达式，而是直接返回运算结果 true。只有当“||”左侧的表达式结果为 false 时，才计算其右侧的表达式。请看下面的例子。

```
public class Test10{
    public static void main(String[] args){
        int a=3,b=4;                        //声明两个变量 a 和 b，并初始化为 3 和 4
        boolean flag;
        flag=(a>2)||(b++>2);                //“||”运算符左侧运算结果为 true，右侧表达式不被运算
        System.out.println("flag="+flag);   //输出结果：flag=true
        System.out.println("b="+b);         //输出结果：b=4
        b=4;                                //重置变量 b 的值
        flag=(a>5)||(b++>2);                //“||”运算符左侧运算结果为 false，右侧表达式被运算
        System.out.println("flag="+flag);   //输出结果：flag=true
        System.out.println("b="+b);         //输出结果：b=5
    }
}
```

所以，逻辑运算符“||”又被称为“简洁或”或“短路或”。而运算符“|”与“||”不同，不管“|”运算符左侧的运算结果是什么，都计算其右侧的表达式，再返回整个逻辑表达式的值。请看下面的例子。

```
public class Test11{
    public static void main(String[] args){
        int a=3,b=4;                        //声明两个变量 a 和 b，并初始化为 3 和 4
        boolean flag;
        flag=(a>2)|(b++>2);                 //“|”运算符左侧运算结果为 true，右侧表达式仍然被运算
        System.out.println("flag="+flag);   //输出结果：flag=true
        System.out.println("b="+b);         //输出结果：b=5
        b=4;                                //重置变量 b 的值
        flag=(a>5)|(b++>2);                 //“|”运算符左侧运算结果为 false，右侧表达式被运算
        System.out.println("flag="+flag);   //输出结果：flag=true
        System.out.println("b="+b);         //输出结果：b=5
    }
}
```



从上例中可以看到，不论“|”运算符左侧的运算结果是什么，其右侧的表达式都会被计算。

3．异或运算符^

当两个关系表达式通过“^”连接在一起时，只要两个关系表达式的值不相同（一个为 true 一个为 false）时，该组合的表达式的值就为真（true）。当运算符两侧的表达式都为 true 或都为 false 时，整个逻辑表达式的值为 false。请看下面的代码。

```
System.out.println(true ^ true);          //输出结果为 false
System.out.println(true ^ false);         //输出结果为 true
System.out.println(false ^ true);         //输出结果为 true
System.out.println(false ^ false);        //输出结果为 false
```

4．取反（非）运算符!

运算符！用于对逻辑值进行取反运算。当逻辑值为 true 时，经过取反运算后，结果为 false；当逻辑值为 false 时，经过取反运算后，结果为 true。请看下面的代码。

```
System.out.println(! true);         //输出结果为 false
System.out.println(!false);         //输出结果为 true
```

2.4.6　位运算符

位运算指的是对操作数以二进制为单位进行的运算，运算结果为整数。也就是说，将操作数转换为二进制表示形式，然后按位进行布尔运算，运算的结果也为二进制。在 Java 语言中，位运算符有 7 个，包括：按位与运算符&、按位或运算符“|”、按位异或运算符“^”、按位取反运算符“～”、左移位运算符“<<”和右移位运算符“>>”和“>>>”。

1．按位与运算符“&”

参与运算的两个数，按位进行与运算，如果两个位都为 1，则该位的运算结果为 1，否则为 0。即：0&0=0，0&1=0，1&0=0，1&1=1。例如下面的代码：

```
System.out.println(6&11);           //输出结果为 2
```

其中，6 的二进制表示为 00000000 00000110，11 的二进制表示为 00000000 00001011。其运算过程如下。

```
    00000000 00000110
&   00000000 00001011
-------------------------
    00000000 00000010      （十进制 2 的二进制表示）
```

所以输出结果为 2。

2．按位或运算符“|”

参与运算的两个数，按位进行或运算，如果两个位都为 0，则该位的运算结果为 0，否则为 1。即：0|0=0，0|1=1，1|0=1，1|1=1。例如下面的代码。

```
System.out.println(6 | 11);         //输出结果为 15
```

其中，6 的二进制表示为 00000000 00000110，11 的二进制表示为 00000000 00001011。其运算过程如下。

```
    00000000 00000110
|   00000000 00001011
-----------------------
    00000000 00001111      （十进制 15 的二进制表示）
```

所以输出结果为 15。

3．按位异或运算符“^”

参与运算的两个数，按位进行异或运算，如果两个位都为 1 或都为 0，则该位的运算结果为 0，否则为 1。即：0^0=0，0^1=1，1^0=1，1^1=0。例如下面的代码。

```
System.out.println(7^11);              //输出结果为 12
```

其中，7 的二进制表示为 00000000 00000111，11 的二进制表示为 00000000 00001011。其运算过程如下。

```
    00000000 00000111
^   00000000 00001011
-----------------------
    00000000 00001100      （十进制 12 的二进制表示）
```

所以输出结果为 12。

4．按位取反运算符“～”

按位取反运算符“～”是一元运算符，它只对一个操作数进行操作。如果是对 1 取反，运算结果为 0；如果是对 0 取反，运算结果为 1，即：～1=0，～0=1。例如下面的代码。

```
System.out.println(13);                //输出结果为-14
```

其中，13 的二进制表示为 00000000 00001101。其运算过程如下。

```
    ～  00000000 00001101
-----------------------
    11111111 11110010      （十进制-14 的二进制表示）
```

所以输出结果为-14。

5．左移位运算符“<<”

左移位运算符“<<”执行一个左移位，即所有二进制位向左移动一位。在作左移位运算时，右边的空位补 0。在不产生溢出的情况下，数据左移一位，相当于乘以 2。例如下面的代码。

```
int a=13;
System.out.println(a<<1);              //输出结果为 26
```

其中，13 的二进制表示为 00000000 00001101。a<<1 的运算过程如下。

```
    00000000 00001101      <<1
-----------------------
    00000000 00011010      （十进制 26 的二进制表示）
```

所以输出结果为 26。



6．右移位运算符“>>”

右移位运算符“>>”执行一个带符号的右移位。在做右移位运算时，左边的空位根据原数的符号位补 0 或 1（原来是负数就补 1，是正数就补 0）。例如下面的代码。

```
int a=-14;
System.out.println(a>>1);            //输出结果为-7
```

其中，-14 的二进制表示为 11111111 11110010。a>>1 的运算过程如下。

```
        11111111 11110010       >>1
        ----------------------------
        11111111 11111001       （十进制-7 的二进制表示）
```

所以输出结果为-7。

右移位运算符“>>>”执行一个不带符号的右移位。在作右移位运算时，左边的空位一律补 0。例如下面的代码。

```
int a=-14;
System.out.println(a>>>1);           //输出结果为 2147483641
```

其中，-14 的二进制表示为 11111111 11110010。a>>>1 的运算过程如下。

```
        11111111 11110010       >>>1
        ----------------------------
        01111111 11111001       （十进制 2147483641 的二进制表示）
```

所以输出结果为 2147483641。

2.4.7　三元运算符

Java 中有一个唯一的三元运算符“?:”，它有三个操作数。这种语法的作用与双分支的选择语句很相似，但其返回值更直接，书写形式更简洁。其语法形式如下。

```
逻辑表达式 ？ 表达式 1 ：表达式 2
```

三元运算符的运算规则是：首先判断逻辑表达式的值，如果为 true，整个三元表达式的值为表达式 1 的值；否则为表达式 2 的值。例如：

```
int score ;
…
System.out.println(score>59? "及格！ " : "不及格！ ");
```

在上例中，首先对表达式“score>59”进行比较。如果代表成绩的变量 score 的值大于 59，那么输出“及格!”；否则输出“不及格!”。

在三元表达式的值，根据对问号（？）前的逻辑表达式的判断，要么取表达式 1 的值，要么取表达式 2 的值，只能两者取其一。例如：

```
int a=14,b=7,c;
c=a>b?++a:++b;
System.out.println(c);          //输出结果为 15
System.out.println(a);          //输出结果为 15
System.out.println(b);          //输出结果为 7
```

2.4.8 运算符的优先级

如果一个表达式中有多个运算符参与混合运算，Java 会根据运算符的优先级别来决定表达式中运算符的执行顺序，也就是说，不同的运算符有不同的优先级。运算符的优先级顺序如表 2.8 所示。

表 2.8 运算符的优先级

优 先 级	描 述	运 算 符	结 合 性
1	分隔符	[] () . , ;	无
2	自增和自减运算、逻辑非	++ -- !	从右到左
3	算术乘除运算	* / %	从左到右
4	算术加减运算	+ -	从左到右
5	移位运算	>> << >>>	从左到右
6	大小关系运算	< <= > >=	从左到右
7	相等关系运算	== !=	从左到右
8	按位与运算	&	从左到右
9	按位异或运算	^	从左到右
10	按位或运算	\|	从左到右
11	逻辑与运算	&&	从左到右
12	逻辑或运算	\|\|	从左到右
13	三元运算	?:	从左到右
14	赋值运算	=	从右到左

例如，下面的表达式中，使用了各种不同优先级的运算符。

```
x>=y && !z 等价于(x>=y) && (!z)
c = a>b?++a:++b 等价于 c = ((a>b)?(++a):(++b))
```

实际上，没有必要去刻意记忆运算符的优先级别。在编写程序时，尽量使用括号来实现想要的运算顺序，以免产生歧义，或使代码难以阅读。

2.5 小结

本章深入学习了 Java 语言的基本语法，重点对常量与变量及数据类型和运算符做了详细的讲述。在本章的学习过程中，可能对 Java 数据类型的转换难以理解，因此读者一定要多做练习，通过上机练习体会和掌握 Java 的基本语法。

通过对本章的学习，读者对 Java 语言的基本语法应有初步的认识，应该掌握变量的含义及如何声明变量，并理解和掌握 Java 的 8 个原始数据类型及其使用。在第 3 章中，将介绍一种重要的引用数据类型——数组。

第3章 数组

在第 2 章的程序中，通过定义单个变量来存储信息，但是在实际应用中，通常有批量的数据需要处理。例如，需要存储 5 个学生的数学成绩。当然，可以在程序中定义 5 个变量来存储，但是再考虑另一个问题：现在程序需要存储 50 个学生的数学成绩，怎么办？当然可以在程序中定义 50 个变量来存储，只不过显得笨拙而已。但是，如果程序需要存储 500 个学生的数学成绩呢？需要存储 5000、50000 个学生的数学成绩呢？



3.1 数组的概念与特点

很显然，在上述情况下，对于同一类型的批量数据，在程序中再通过定义单个的变量这种方式已经不能解决存储大批量数据的问题了，因为它既不经济，也不实际。对于这种类型的问题，Java 提供了“数组”这种引用数据类型来解决。

3.1.1 什么是数组

数组是用来存储具有相同数据类型的数据的集合，可以使用共同的名字来引用数组中存储的数据。数组可以存储任何类型的数据，包括原始数据类型和对象，但是一旦指定了数组的类型之后，就只能用来存储指定类型的数据。

例如，可以在程序中定义 5 个变量来存储 5 个学生的数学成绩，这些成绩都是单精度浮点类型，使用如下代码。

```
float mathScore_1 = 73.5F;
float mathScore_2 = 82.0F;
float mathScore_3 = 95.0F;
float mathScore_4 = 63.0F;
float mathScore_5 = 75.5F;
```

很显然，如果要存储 500 个、5000 个学生的数学成绩的话，这样就不能满足需要了。这时就需要使用数组来存储同一类型大批量的数据了。

可以在程序中定义一个只能用来存储整数的整型数组，也可以定义一个只能用来存储小数的 float 类型的数组，或者一个只能用来存储字符串的 String 对象数组，甚至于一个只能用来存储某一类型数组的数组。但是，不可以定义一个既能存储整数，又能存储小数或

其他类型数据的数组。

数组提供了一种数据分组的便利方法。在数组中，每一项数据都经过编号，可以通过数组的下标（即数据项在数组中的索引值，从 0 开始。例如，数组中第一项数据的下标为 0，第二项数据的下标为 1，第 N 项数据的下标为 $N-1$）来方便地访问。

3.1.2 Java 语言中数组的特点

在不同的程序设计语言中，一般都提供有数组这种数据类型，功能和使用方法都很类似。不过 Java 语言中的数组有自己的几个特点：

- Java 中的数组，既能存储原始数据类型，又能存储对象类型。
- 数组元素的个数称为数组的长度。数组长度一旦确定，就不能改变。
- 数组元素的下标是从 0 开始的，即第一个元素的下标是 0。
- Java 不支持多维数组，但是可以创建数组的数组。
- Java 中将数组作为对象处理。数组对象含有成员变量 length，用来表示数组的长度。

在本章的后面的内容中，将对这些特点逐一做详细讲解。

3.2 一维数组

在编写程序时，最经常用到的是一维数组。要在 Java 里创建并使用一维数组，需要以下步骤。

1. 声明一个数组变量来存放该数组。
2. 创建一个新的数组对象并将其赋值给上一步所创建的数组变量。
3. 在该数组中存储数据。
4. 访问或修改此数组中存储的数据（元素）。

3.2.1 声明一维数组变量

在创建和使用数组过程中，首先要做的就是声明一个数组变量，来保存实际的数组。声明数组变量的含义如同声明一个原始数据类型的变量，要指明数组将要容纳的数据类型及数组的名字。声明一维数组的语法如下。

```
数组类型[ ] 数组名字;
数组类型 数组名字[ ];
```

这两种数组变量声明的方式在作用上是相同的，但通常建议采用第一种方式，它更符合数组变量的原理，即在数组名字前面指定数组可以保存的数据类型，这里使用方括号“[]”来代表数组类型。例如，可以分别声明一个 int 型、一个 boolean 型、一个 float 型的一维数组。

```
int[ ]  weeks;                          //声明一个整型数组
boolean[ ]  flag;                       //声明一个布尔类型的数组
float[ ]  score;                        //声明一个浮点类型的数组
```

也可以使用第二种声明数组变量的方式。

```
int  weeks[ ];
boolean  flag[ ];
float  score[ ];
```



3.2.2　创建一维数组对象

声明了数组变量之后，下一步就需要创建一个数组对象，并将其赋给所声明的数组变量。而创建数组对象，实际上就是在内存中分配相应大小的存储空间给数组。有两种方式可以用来创建数组对象。

1．使用 new 运算符

在 Java 中，数组是引用对象，所以可以使用 new 运算符来直接创建一个数组对象，如以下代码所示。

```
int[ ]  weeks = new int[ 7] ;                  //创建一个整型数组对象，数组大小为 7
String[ ] studentNames = new  String[10];   //创建一个字符串数组，数组大小为 10
```

使用 new 运算符来创建数组对象时，必须指定这个数组的大小。创建数组对象时，仅仅是在内存中为数组变量分配指定大小的空间，并没有实际存储数据，这时数组的所有元素会被自动地赋予初值，其中：

- 数字数组，初值是 0。
- 布尔数组，初值是 false。
- 字符数组，初值是‘\0’。
- 对象数组，初值是 null。

2．直接初始化数组元素

也可以通过把数组元素值直接放在大括号中的办法来同时完成创建和初始化数组，这时将在内存中分配与元素数量相适应的空间大小，并赋予指定的元素值，如以下代码所示。

```
int[ ]  weeks = {1,2,3,4,5,6,7} ;                    //创建一个整型数组对象，数组大小为 7，并同时赋初值
String[ ] studentNames = {"张小明","黄芳","赵秀丽"}; //创建一个 String 数组对象，并初始化
```

使用这种方法创建数组对象时，大括号里的元素类型必须与指定的数据类型一致，并且数组的大小与大括号中包含的元素个数相同。

3.2.3　访问一维数组元素

当创建了数组变量并赋予数组变量带有初值的数组对象以后，就可以访问一维数组中的元素了。访问一维数组元素，使用该数组的名字，后跟数组下标，如以下代码所示。

```
String[ ] studentNames = new  String[10];
studentNames[0] = "刘金波";                          //访问数组中的第一个元素
```

这里将数组 studentNames 的第一个元素 studentNames[0]的值设置为字符串“刘金波”。一定要注意，在 Java 的数组中，下标是从 0 开始计数的，所以长度为 n 的数组第一个元素的下标为 0，最后一个元素的下标为 $n-1$。再看下面这个示例。

```
public class Test1{
    public static void main(String[ ] args){
        int age[ ] = new int[2];            //定义一个整型数组，大小为 2
        String names[ ] = new String[3];    //定义一个字符串类型数组，大小为 3
        System.out.println(age[0]);         //输出数组 age 的第一个元素值
```

```
        System.out.println(age[1]);          //输出数组 age 的第二个元素值
        //System.out.println(age[2]);        //错误，访问超出了数组的范围
        System.out.println(names[0]);        //输出数组 names 的第一个元素值
        System.out.println(names[1]);        //输出数组 names 的第二个元素值
        System.out.println(names[2]);        //输出数组 names 的第三个元素值
        //System.out.println(names[3]);      //错误，访问超出了数组的范围
    }
}
```

在本示例中，创建数组对象时，自动赋予初值。数组元素的下标从 0 开始。Java 会严格检查数组的边界，以防止引用超出数组范围的值。Java 运行时环境会检查数组下标以确保所有的数组下标都在正确的范围之内。在这里，Java 运行时环境将会检查 age 数组的每个下标的值，确保它在 0～2 之间；运行时环境会检查 names 数组的每个下标的值，确保它在 0～3 之间。如果在程序中引用了下标超出数组范围的元素，Java 编译器就会显示一个“ArrayIndexOutOfBoundsException”错误提示。

所以在 Java 中，无法对一个超出数组范围的数组元素进行访问或赋值，这就避免了在其他语言中因为数组越界所引起的一系列问题。上例输出结果为：

```
0
0
null
null
null
```

读者可以将上例中注释掉的语句前面的注释符号去掉，试着编译一下，会看到相应的错误提示。将数组越界检查放在编译时进行检查，可以避免运行时出现难以排查的错误。也可以在创建数组的同时，赋值进行初始化。请看下面这个示例。

```
public class Test2{
    public static void main(String[ ] args){
        int age[ ] = {18,23};          //定义一个整型数组，大小为 2，并赋值初始化
        System.out.println(age[0]);      //输出数组 age 的第一个元素值
        System.out.println(age[1]);      //输出数组 age 的第二个元素值
        String names[ ] =  {"张小明","黄芳","赵秀丽"}; //定义一个字符串类型数组，大小为 3，并赋值初始化
        System.out.println(names[0]);    //输出数组 names 的第一个元素值
        System.out.println(names[1]);    //输出数组 names 的第二个元素值
        System.out.println(names[2]);    //输出数组 names 的第三个元素值
    }
}
```

在本示例中，创建数组对象时，直接初始化数组元素。输出结果为：

```
18
23
张小明
黄芳
赵秀丽
```

3.2.4 修改一维数组元素

数组的大小不可改变，但数组中的元素值是可以改变的。在声明一个数组变量和创建一个



数组对象以后，可以通过为数组中的元素赋值，来修改数组中任一元素的值。请看下面的代码。

```
public class Test3{
    public static void main(String[ ] args){
        int mathScore[ ] = {78,93,84};       //定义一个整型数组，用来保存三个同学的数学成绩
        int englishScore[ ] = new int[3];    //定义一个整型数组，用来保存三个同学的英语成绩
        int totalScore[ ] = new int[3];   //定义一个数组，用来保存三个同学的总成绩
        String[ ] students = {"张小明","黄芳","赵秀丽"};
        //为三个同学的英语成绩赋值
        englishScore[0] = 98;                //为第一个元素赋值
        englishScore[1] = 72;                //为第二个元素赋值
        englishScore[2] = 65;                //为第三个元素赋值
        totalScore[0] = mathScore[0] + englishScore[0];     //计算第一个同学的总成绩
        totalScore[1] = mathScore[1] + englishScore[1];     //计算第二个同学的总成绩
        totalScore[2] = mathScore[2] + englishScore[2];     //计算第三个同学的总成绩
        //输出三个同学的总成绩
        System.out.println(students[0] + "同学的总成绩为: " + totalScore[0]);
        System.out.println(students[1] + "同学的总成绩为: " + totalScore[1]);
        System.out.println(students[2] + "同学的总成绩为: " + totalScore[2]);
    }
}
```

在这个示例中，先声明 4 个数组，其中 mathScore 和 englishScore 存储学生的数学和英语成绩，students 存储学生的姓名，totalScore 存储学生的总成绩。通过赋值的方式为数组 englishScore 初始化成绩。访问 mathScore 和 englishScore 数组中对应下标的元素，相加以后赋值给 totalScore 数组相应的元素，最后输出三个同学的姓名及其总成绩，如下所示。

```
张小明同学的总成绩为：176
黄芳同学的总成绩为：165
赵秀丽同学的总成绩为：149
```

通常数组的大小会远远大于本章示例中的数组大小，所以在对数组进行操作时，一般都使用循环来重复一个数组元素，而不需要一个一个单独地处理。关于循环结构，会在第 4 章进行讲解。



3.3　多维数组

数组元素除了可以是原始数据类型、对象类型之外，还可以是数组，即数组的元素是数组。所以虽然 Java 不支持多维数组，但是可以通过声明数组的数组来实现同样的功能。元素数组还可以包含数组，依此类推，最后创建出达到需要的任意维数。

3.3.1　声明二维数组变量

在实际应用中，多维数组中使用最多的是二维数组。所以本节主要介绍二维数组的声明、创建和使用，对于其他多维数组的声明、创建和使用，与二维数组相似，只做简单说

明。声明二维数组的语法如下。

```
数组类型[ ][ ] 数组名字;
数组类型 数组名字[ ][ ];
```

这两种数组变量声明的方式在作用上是相同的，但通常建议采用第一种方式，与声明一维数组类似。例如，可以分别声明一个 int 型、一个 boolean 型、一个 float 型的二维数组，代码如下。

```
int[ ][ ]  year;
boolean[ ][ ]  flag;
float[ ][ ]  score;
```

也可以把括号放在数组名称后面，如下所示。

```
int  year[ ][ ];
boolean  flag[ ][ ];
float  score[ ][ ];
```

对于二维数组是这样，那么对于其他多维数组同样如此。例如，以下代码定义了一个三维数组。

```
int[ ][ ][ ]  threeDimension;
```

3.3.2 创建二维数组对象

声明了数组变量之后，下一步就需要创建一个数组对象，并将其赋给所声明的数组变量。而创建数组对象实际上就是在内存中分配相应大小的存储空间给数组。有两种方式可以用来创建二维数组对象。

1. 使用 new 运算符

在 Java 中，数组是引用对象，所以可以使用 new 运算符来直接创建一个数组对象，如以下代码所示。

```
int[ ][ ]  year = new int[52][7] ;
```

使用 new 运算符来创建二维数组对象时，必须指定这个数组的大小。在这个代码中，创建了一个二维数组对象，用来保存一年中的 52 个星期及每个星期的 7 天。

可以把二维数组想象成一个二维表格，带有行和列。其中行与二维数组的第一维对应，列与二维数组的第二维对应。例如，对于下面的代码。

```
int[ ][ ]  arrays1= new int[3][5] ;
```

可以把数组 arrays1 看成是如表 3.1 所示的二维表格。对于二维数组来说，第一个下标代表着所在的行，第二个下标代表着所在的列。下标也是从 0 开始计算的。

表 3.1　二维数组结构

	列下标 0	列下标 1	列下标 2	列下标 3	列下标 4
行下标 0	arrays[0][0]	arrays[0][1]	arrays[0][2]	arrays[0][3]	arrays[0][4]
行下标 1	arrays[1][0]	arrays[1][1]	arrays[1][2]	arrays[1][3]	arrays[1][4]
行下标 2	arrays[2][0]	arrays[2][1]	arrays[2][2]	arrays[2][3]	arrays[2][4]



2．直接初始化数组元素

也可以通过把数组元素值直接放在大括号中的办法，来同时完成创建和初始化二维数组。在大括号中使用逗号分隔每个花括号，每个花括号中为用逗号分开的行数据。这时将在内存中分配与元素数量相适应的空间大小，并赋予指定的元素值，如以下代码所示。

```
int[ ][ ]  arrays2 = {{11,12,13,14,15},{21,22,23,24,25},{31,32,33,34,35}};
```

在这里，定义了一个二维数组 arrays2，它有 3 行 5 列。可以把数组 arrays2 看成如表 3.2 所示的二维表格。

表 3.2　二维数组 arrays2 的结构

	列下标 0	列下标 1	列下标 2	列下标 3	列下标 4
行下标 0	11	12	13	14	15
行下标 1	21	22	23	24	25
行下标 2	31	32	33	34	35

使用这种方式创建二维数组，数组中每一行的长度也可以不同，因为可以把二维数组看成是元素为一维数组的一维数组，所以元素一维数组的长度可以不同。例如：

```
int[ ][ ]  arrays2 = {{11,12 },{21,22,23,24,25},{31, 34,35}};
boolean flags = {{true,false},{true,true,false,false},{false,false,true}};
```

3.3.3　访问二维数组元素

访问二维数组元素，指的是读取二维数组中指定位置的元素。要访问二维数组元素，需使用该数组的名字，后面跟两个中括号代表下标，第一个下标为行索引，第二个下标为列索引。如下代码所示。

```
int[ ][ ]  arrays2 = {{11,12,13,14,15},{21,22,23,24,25},{31,32,33,34,35}};
System.out.println(arrays2[0][2]);
```

上述代码输出数组 arrays2 的第 1 行（下标为 0）第 3 列（下标为 2）的元素。从表 3.2 中可以看出，应该输出的值为 13。



在二维数组中，行和列的下标都是从 0 开始计数的。

3.3.4　修改二维数组元素

在声明一个数组变量和创建一个二维数组对象以后，也可以通过为数组中的元素赋值，来修改数组中任一元素的值。这与修改一维数组很类似。请看下面的代码。

```
public class Test4{
    public static void main(String[ ] args){
        int[ ][ ]  arrays = {{11,12,13,14,15},{21,22,23,24,25},{31,32,33,
34,35}};                                                    //定义二维数组 arrays
        arrays[0][0] = arrays[1][3] + arrays[2][0]; //将两个数组元素相加
        System.out.println(arrays[0][0]);           //输出结果为 55
    }
}
```

在这个示例中，读取数组第 2 行第 4 列（行和列的下标都是从 0 开始计算）的元素值

24，与数组第 3 行第 1 列的元素值 31 相加，并将结果 55 赋给数组第一行第一列的元素 arrays[0][0]，最后输出 arrays[0[0]的值 55。

通常在对二维数组进行操作时，一般都使用双重循环来处理。关于循环结构，会在第 4 章进行讲解。



3.4 小结

本章深入学习了 Java 语言的数组，了解了 Java 语言中数组的概念，学习了如何声明、创建、访问和修改数组。本章重点是一维数组和二维数组。其中对于数组的创建及二维数组的创建和使用是本章学习的难点，建议读者多上机练习。在学习了后续的循环语句后，会进一步向读者介绍对于二维数组的高级操作。在第 4 章将向读者介绍基本的程序流程控制语句。

第4章　程序流程控制语句

Java 语言中的程序流程控制语句有三种：顺序结构、选择结构和循环结构。其中顺序结构最简单，程序依次执行各条语句。Java 的每条语句一般以分号（“；”）作为结束标志。所以本章重点讲解选择结构和循环结构。



4.1　选择语句

在程序设计时，经常需要使用选择结构在程序中完成逻辑判断和选择功能，这就需要使用到选择语句。Java 中的选择语句包括 if 语句、if-else 语句和 switch 语句。选择语句用来控制选择结构，对选择条件进行判断，并根据判断结果选择要执行的程序语句，改变程序执行流程。

4.1.1　if 选择语句

在实际生活中，经常会需要做一些逻辑判断，并根据逻辑判断结果做出选择。例如，当一个男孩子向一个非常理智的女孩子求婚时，这个女孩子思考的逻辑可能是这样的：

```
如果你有 100 万
    那么，我就嫁给你!
```

在这里，女孩子嫁给男孩子这个过程的实施，必须以“你有 100 万”为前提。只有逻辑判断“你有 100 万”这一前提条件为真时，“那么，我就嫁给你!”这个行为才能得到执行。在生活当中，类似这样的情形是很常见的。

相应地，在 Java 程序设计语言中，也有相应的条件语句来完成类似的逻辑判断和有选择地执行这样的功能，这就是 if 语句。if 语句的语法格式如下。

```
if(条件表达式)
    语句
```

或者

```
if(条件表达式){
    一条或多条语句
}
```

if 语句执行的过程如下。

（1）对 if 后面括号里的条件表达式进行判断。

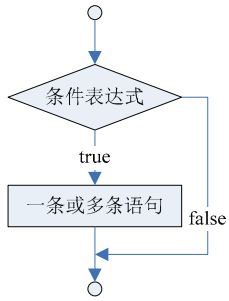


图 4.1　if 语句流程图

（2）如果条件表达式的值为 true，就执行表达式后面的语句或后面大括号里的多条语句。

（3）如果条件表达式的值为 false，则跳过 if 语句，执行下一条语句。

if 语句是只有一个选择的语句结构，所以又叫单分支选择结构，其流程图如图 4.1 所示。

需要注意的是，在条件表达式的右括号后面，如果只有一条执行语句的话，那么可以跟一对大括号，也可以不跟大括号。如果有多条语句需要一起执行，则必须用大括号把多条语句括起来，形成语句块。建议不论条件成立时后面要执行多少条语句，一律用大括号括起来。请看下面的代码。

```
int a = 2,b = 3;
if(a > b)
    System.out.println("这句话能输出！");
    System.out.println("这句话也能输出！");        //这个输出语句并不是条件表达式后的执行语句
```

因为没有用大括号将条件表达式 a>b 后的两个输出语句括起来，所以条件表达式成立与否，只影响到其后第一个输出语句。这段代码的输出结果为：

```
"这句话也能输出！"
```

用 if 语句表示女孩子考虑求婚问题，可以以自然语法写成如下形式：

```
if(男朋友有 100 万){
    我愿意嫁给你!
}
```

当然，在程序中，需要使用 Java 语言的语法来表示，如下所示。

```
float boyFriendMoney;                    //变量 boyFriendMoney 代表男朋友的存款
...
if(boyFriendMoney > 100){                //如果男朋友的存款多于 100 万
    System.out.println("我愿意嫁给你！");//那么，同意嫁给他
}
```

4.1.2　if-else 双分支选择语句

有时候，需要根据逻辑判断结果分别做出选择。例如，在上一节的例子中，当一个男孩子向一个女孩子求婚时，女孩子思考的逻辑也可能是这样的：

```
如果你有 100 万
    那么，我就嫁给你!
否则
    我就拒绝嫁给你。
```

在这里，女孩子根据对方是否有 100 万，有两个选择：当逻辑判断“你有 100 万”这一前提条件为真时，执行“那么，我就嫁给你!”这个行为；否则，如果逻辑判断不为真，就执行“我就拒绝嫁给你”。在生活当中，类似这样的情形是很常见的。



相应地，在 Java 程序设计语言中，也有相应的条件语句来完成类似的逻辑判断和有选择地执行这样的功能，这就是 if-else 语句。if-else 语句的语法格式如下。

```
if(条件表达式)
    语句 1
else
    语句 2
```

或者

```
if(条件表达式){
    语句块 1
}else{
    语句块 2
}
```

if-else 语句执行的过程如下。

（1）对 if 后面括号里的条件表达式进行判断。

（2）如果条件表达式的值为 true，就执行语句 1 或语句块 1。

（3）如果条件表达式的值为 false，就执行语句 2 或语句块 2。

if-else 语句又称为双分支选择结构，其流程图如图 4.2 所示。

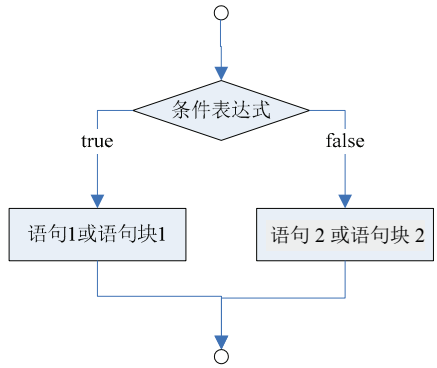


图 4.2　if-else 语句流程图

【例 4.1】编写一个程序，求两个数中比较大的值，并输出。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 MaxValueDemo.java。

```
public class MaxValueDemo{
    public static void main(String[] args){
        int a=23,b=36,max;                    //声明三个整型变量
        if(a>b){                              //如果 a 大于 b
            max = a;                          //将较大值的 a 赋给变量 max
        }else{                                //否则，如果 a 不大于 b
            max = b;                              //将 b 的值赋给变量 max
        }
        System.out.println(“最大值是:” + max);     //输出变量 max 的值，即输出最大值
    }
}
```

② 使用如下命令编译和运行 MaxValueDemo.java 源程序。

```
javac MaxValueDemo.java
java MaxValueDemo
```

③ 程序输出结果如下。

```
最大值是：36
```

在程序 MaxValueDemo 中，定义三个整型变量 a、b 和 max。对变量 a 和 b 的值进行判断，并将其中较大的值赋给变量 max，最后输出 max 的值。

4.1.3　if-else-if 多分支选择语句

除了单分支结构的 if 语句和双分支结构的 if-else 语句之外，还有一种常用的选择结构

是多分支结构，使用 if-else-if 语句实现。if-else-if 语句的语法格式如下。

```
if(条件表达式 1){                          //如果条件表达式 1 成立（结果为 true）
    语句块 1                               //就执行语句块 1 中的代码
}else if(条件表达式 2){                    //否则，如果条件表达式 2 成立
    语句块 2                               //就执行语句块 2 中的代码
}
…                                          //对其他条件进行判断
else if(条件表达式 n-1){                   //如果条件表达式 n-1 成立
    语句块 n-1                             //就执行语句块 n-1 中的代码
}else{                                     //如果以上所有的条件都不成立
    语句块 n                               //就执行语句块 n
}
```

if-else-if 语句执行的过程如下。

（1）对 if 后面括号里的条件表达式进行判断。

（2）如果条件表达式的值为 true，就执行语句块 1。

（3）否则，对条件表达式 2 进行判断。如果条件表达式的值为 true，就执行语句块 2。

（4）否则，依此类推。

（5）如果所有条件表达式的值都为 false，最后执行语句块 n。

if-else-if 语句依次对 if 后面的条件表达式进行判断，遇到第一个值为真的表达式时，就执行其后面的语句块，然后整个 if-else-if 语句就结束了，不再对后面的条件表达式进行判断和执行。理论上，可以有无限多个 else if 子句。

【例 4.2】编写一个程序，根据学生的成绩，判断其属于哪个档次，并输出。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 ScoreDemo.java。

```
public class ScoreDemo{
    public static void main(String[] args){
        int score = 85;          //定义代表学生成绩的变量 score，并赋初值
        if(score >= 90){         //如果成绩大于等于 90 分
            System.out.println("您的成绩优秀!");
                                 //执行这个语句块，并结束本 if-else-if 语句
        }else if(score >= 80){   //满足这个条件表达式
            System.out.println(“您的成绩良好”);
                                 //执行这个语句块，并结束本 if-else-if 语句
        }else if(score >= 70){   //如果成绩大于等于 70 分
            System.out.println(“您的成绩中等”);
                                 //执行这个语句块，并结束本 if-else-if 语句
        }else if(score >= 60){   //如果成绩大于等于 60 分
            System.out.println(“您的成绩及格”);
                                 //执行这个语句块，并结束本 if-else-if 语句
        }else{                   //如果以上条件都不成立，即成绩小于 60 分
            System.out.println(“您的成绩不及格”);
        }                        //结束 if 语句是直接转到这里，执行 if 后面的语句
    }
}
```

② 使用如下命令编译和运行 ScoreDemo.java 源程序。

```
javac ScoreDemo.java
java ScoreDemo
```

③ 程序输出结果如下。

```
您的成绩良好
```

在本例中，当程序执行到 score >= 80 这个表达式时，计算其结果为 true，执行其后的语句块，输出“您的成绩良好”，并结束 if-else-if 语句，转到该语句最后一个大括号后面，执行其后面的语句。if-else-if 语句又称为多分支选择结构，其流程图如图 4.3 所示。

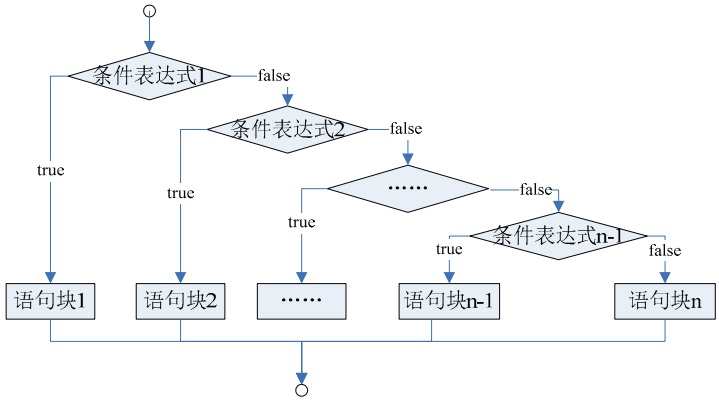


图 4.3　if-else-if 语句流程图

需要注意的是，最后的 else 语句可以省略，那么当所有的条件表达式都不成立时，什么也不执行。

【例 4.3】编写一个程序，求三个整数中的最大值，并输出。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 MaxValueDemo2.java。

```
public class MaxValueDemo2{
    public static void main(String[] args){
        int a=23,b=36,c=25,max;          // 声明 4 个整型变量，其中 a,b,c 赋初值
        if(a>c){                         // 如果 a 比 c 大
            max = a;                     // 那么把 a 赋给 max 变量
        }else if(b>c){                   // 如果 a 比 c 小，那么 b 与 c 进行比较。如
果 b 大于 c
            max = b;                     // 那么最大值为 b
        }else                            // 否则，a 和 b 都不大于 c
            max = c;                     // 那么最大值为 c
        System.out.println(“最大值是:” + max); // 输出最大值 max 的值
    }
}
```

② 使用如下命令编译和运行 MaxValueDemo2.java 源程序。

```
javac MaxValueDemo2.java
java MaxValueDemo2
```

③ 程序输出结果如下。

```
最大值是：36
```

在程序 MaxValueDemo2 中，对三个整数 a、b 和 c 进行比较。实际上，这个程序逻辑

并不完整，例如，将变量 a 的值改为 26，再编译并运行此程序，会发现输出的最大值为 26，显然是不正确的。这是因为没有考虑 a 大于 c 但小于 b 的情况。这时，需要使用嵌套的 if 语句。

4.1.4 选择语句的嵌套

if 语句里面可以再嵌套 if 语句。所谓嵌套，就是在 if 语句中又包含一个或多个 if 语句。这样的语句一般用在比较复杂的分支语句中。其语法格式如下。

```
if(条件表达式 1){
    if(条件表达式 2){
        语句块 1
    }else{
        语句块 2
    }
}else{
    if(条件表达式 3){
        语句块 3
    }else{
        语句块 4
    }
}
```

【例 4.4】使用嵌套的 if 语句编写一个程序，根据学生的成绩，判断其属于哪个档次，并输出。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 ScoreDemo2.java。

```
public class ScoreDemo2{
    public static void main(String[] args){
        int score = 85;                    // 声明初始整型变量 score，并赋初值
        if(score >=60){                    // 如果成绩大于等于 60，再进一步判断
            if(score >= 70){               // 如果成绩大于等于 70，再进一步判断
                if(score >= 80){           // 如果成绩大于等于 80，再进一步判断
                    if(score >= 90)        // 如果成绩大于等于 90，输出相关信息
                        System.out.println("您的成绩优秀!");
                    else                   // 如果成绩在 80～90 之间，输出相关信息
                        System.out.println("您的成绩良好");
                }else{                     // 如果成绩在 70～80 之间，输出相关信息
                    System.out.println("您的成绩中等");
                }
            }else{                         // 如果成绩在 60～70 之间，输出相关信息
                System.out.println("您的成绩及格");
            }
        }else{                             // 如果成绩小于 60
            System.out.println("您的成绩不及格");
        }
    }
}
```

② 使用如下命令编译和运行 ScoreDemo2.java 源程序。

```
javac ScoreDemo2.java
java ScoreDemo2
```



③ 程序输出结果如下。

```
您的成绩良好
```

同样求三个整数中的最大值，也可以使用嵌套的 if-else 语句。请看下面的示例。

【例 4.5】使用嵌套的 if-else 语句编写一个程序，求三个整数中的最大值，并输出。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 MaxValueDemo3.java。

```
public class MaxValueDemo3{
    public static void main(String[] args){
        int a=23,b=36,c=25,max;                    //声明三个整型变量
        if(a>c){                                   //如果 a>c，最大值在 a 和 b 中间
            if(a>b)
                max = a;
            else
                max = b;
        }else{                                     //如果 a<c，最大值在 b 和 c 中间
            if(b>c)
                max = b;
            else
                max = c;
        }
        System.out.println("最大值是:" + max);
    }
}
```

② 使用如下的命令编译和运行 MaxValueDemo3.java 源程序。

```
javac MaxValueDemo3.java
java MaxValueDemo3
```

③ 程序输出结果如下。

```
最大值是：36
```



else 总是和离它最近的 if 进行匹配，所以在 if 语句嵌套时，尽可能使用大括号划分逻辑关系，避免出现问题。例如，上例也可以这样来编码：

```
public class Test{
    public static void main(String[] args){
        int score = 85;      //声明整型变量，并赋初值
        if(score >=60){ //如果变量 score（成绩）的值大于等于 60，再做进一步的判断
            if(score > 89)  //如果变量 score（成绩）的值大于 89，输出相关的信息
                System.out.println("恭喜您！您的得分是优秀!");
        }else               //如果成绩小于 60，输出相关信息
            System.out.println("很抱歉，您考试不及格!");
    }
}
```

上面代码中第一个 if 语句后面的大括号是必需的，这时什么也不输出。如果去掉第一个 if 语句后面的大括号，那么，else 子句将与它最近的那个 if 语句匹配，这时代码如下所示。

```
public class Test{
```

```
    public static void main(String[] args){
        int score = 85;
        if(score >=60)
            if(score > 89)
                System.out.println(“恭喜您！您的得分是优秀!”);
        else        //与最近的 if 语句相匹配，因此当 score 小于等于 89 时，输出信息
            System.out.println(“很抱歉，您考试不及格!”);
    }
}
```

上面代码的输出结果如下。

```
“很抱歉，您考试不及格！”
```

很明显，输出结果不是所期望的。因此建议，为了避免这种情况，最好的方法是加上大括号为代码划分界限。

4.1.5 switch 多分支开关语句

当选择结构的分支越多时，if-else-if 语句就会变得越来越难懂。Java 提供了另一种多分支语句——switch 语句。switch 语句是多分支的开关语句。它的语法格式如下。

```
switch(表达式){
    case 常量表达式 1：语句组 1；
                    [ break;]
    case 常量表达式 1：语句组 1；
                    [ break;]
    …
    case 常量表达式 1：语句组 1；
                    [ break;]
    default：语句块 n
}
```

说明：

- switch 后面的表达式值的数据类型可以是字符型（char）、字节型（byte）、短整型（short）或者整型（int），但不可以是布尔型（boolean）、长整型（long）、浮点型（float、double）。
- case 后面的常量表达式的值的类型，必须与 switch 后面的表达式值的类型相匹配，而且必须是常量表达式或直接字面量。
- break 语句可以省略。如果省略，那么程序会按顺序执行 switch 中的每一条语句，直到遇到右大括号或 break 为止。
- 语句块可以是一条语句或多条语句，但不需要使用大括号括起来。
- case 分支语句和 default 语句都不是必需的，可以省略。

switch 语句执行的过程如下。

（1）将 switch 表达式的值与各个 case 后面的常量表达式的值一一进行比较。

（2）当表达式的值与某个 case 分支的值相等时，程序执行从这个 case 分支开始的语句组。

（3）如果没有任何一个 case 分支的值与 switch 表达式的值相匹配，并且 switch 语句含有 default 分支语句，则程序执行 default 分支中的语句组。

（4）直到遇到 break 语句或右大括号，结束 switch 语句。

switch 语句的执行流程图如图 4.4 所示。

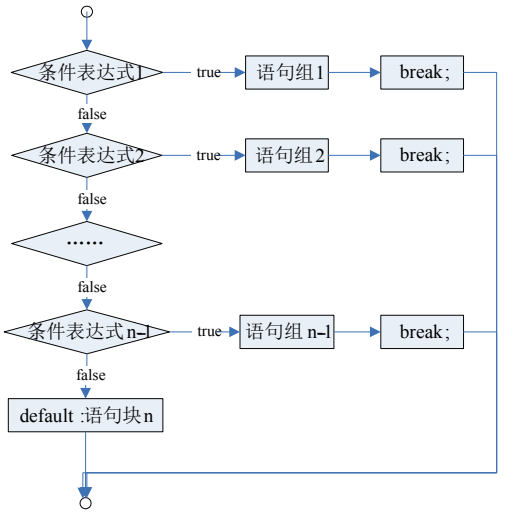


图 4.4　switch 语句流程图

【例 4.6】使用 switch 语句编写一个程序，要求根据学生成绩等级，输出成绩为“优秀”、“良好”、“中等”、“及格”和“不及格”。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 SwitchDemo.java。

```
public class SwitchDemo{
    public static void main(String[] args){
        char studentGrade = 'B';
            //声明字符变量，并赋初值
        switch(studentGrade){
            //匹配 studentGrade 的值
            case 'A':                   //如果 studentGrade 的值与'A'或'a'相符
            case 'a':
                System.out.println("优秀");
                break;                  //结束 switch 语句
            case 'B':                   //如果 studentGrade 的值与'B'或'b'相符
            case 'b':
                System.out.println("良好");
                break;                  //结束 switch 语句
            case 'C':                   //如果 studentGrade 的值与'C'或'c'相符
            case 'c':
                System.out.println("中等");
                break;                  //结束 switch 语句
            case 'D':                   //如果 studentGrade 的值与'D'或'd'相符
            case 'd':
                System.out.println("及格");
                break;                  //结束 switch 语句
            default:                    //如果以上条件都不相符
                System.out.println("不及格");
        }
    }
}
```

② 使用如下命令编译和运行 SwitchDemo.java 源程序。

```
javac SwitchDemo.java
java SwitchDemo
```

③ 程序输出结果如下。

```
良好
```

在这个例子中，执行 switch 语句时，将学生的分数 studentGrade 依次与 case 后面的字符常量进行比较，当分数为字符‘B’时，执行其后面的语句。由于“case ‘B’”后面为空语句，程序会按顺序执行，进而执行“case ‘b’”后面的语句，即输出字符串“良好”。再往下执行，

遇到 break 语句，程序结束 switch 语句，跳转到 switch 语句的右大括号处，switch 语句结束。

4.1.6 if 选择语句与 switch 开关语句的区别

从功能上来讲，if 语句和 switch 语句都是多分支选择语句，在通常情况下，对于多分支选择结构，使用 if 语句和使用 switch 语句从作用上讲是相同的。但是在实际编写程序时，一般遵循下面的使用原则。

（1）如果分支的层次不超过三层，那么通常使用 if-else-if 语句；否则，使用 switch 语句。

（2）如果条件判断语句是对一个变量是否属于一个范围进行判断，如“a>60 && a<89”，这时要使用 if-else-if 语句。

（3）如果是对同一个变量的不同值做条件判断，既可以使用 if-else-if 语句，也可以使用 switch 语句。但建议优先使用 switch 语句，其执行效率相对高一些。

4.2 循环语句

一个循环是语句或语句集重复执行的任何一段代码块，它重复执行同一套指令直到一个结束条件出现。Java 中的循环语句包括：

- while 语句；
- do-while 语句；
- for 语句。

4.2.1 while 循环语句

在第 3 章“数组”中，为数组元素赋值都是逐个赋值的，这在数组比较小的时候是合适的。但是，如果数组的大小足够大时，这种逐个赋值的方法就不切实际了。例如下面代码：

```
int[ ]  mathScore = new int[200];
```

声明一个整型数组 mathScore，用来存储学生的数学成绩，大小为 200。如果要为这个数组中的每个元素逐个赋值，效率是极为低下的，操作重复且代码量大。所以，对于这样重复性的操作，Java 语言提供了循环语句来解决，包括 while 语句、do-while 语句和 for 语句。其中 while 语句的基本语法格式如下。

```
while(布尔表达式){
    语句或语句块
}
```

说明：

- 表达式可以是而且必须是任何运算结果为布尔值的表达式。
- 如果只有一条语句的话，可以省略表达式后面的大括号。但建议不论几条语句，都使用大括号括起来。
- while（表达式）后面一定没有分号（;），在右大括号后面也没有分号（;），这是初学者经常容易犯的错误。

while 循环语句的执行过程如下。

（1）首先判断 while 后面小括号中表达式的值，如果为 true，就从后面紧跟的左大括



号开始，按顺序执行大括号里的语句，这称为“执行循环体”。

（2）循环体执行结束，再次返回对 while 后面小括号中的表达式的值进行判断，重复执行上一步，直到表达式的值为 false。

（3）此时结束执行 while 语句。while 循环结束。

【例 4.7】使用 while 语句编写一个程序。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 WhileDemo1.java。

```
public class WhileDemo1{
    public static void main(String[] args){
        int[ ] mathScore = new int[200];
        //为数组中的所有元素赋初值 60
        int i = 0;                          //定义循环变量
        while(i<200){                       //循环 200 遍
            mathScore[i] = 60;              //为每一个数组元素赋初值 60
            i++;                            //循环变量自增 1
        }
    }
}
```

② 使用如下命令编译和运行 SwitchDemo1.java 源程序。

```
javac SwitchDemo1.java
java SwitchDemo1
```

在上面这个示例中，首先定义了一个循环变量 i，并赋初值为 0。循环变量 i 用在 while 后面的表达式 i<200 中，用来作为循环的条件，保证循环只重复 200 次，同时作为数组元素引用的下标。在循环体语句中，一定要有改变循环条件的语句 i++。这样每执行一次循环体，变量 i 自增 1，当循环执行 200 遍时，i 的值为 200，循环条件不再成立，结束循环。

改变循环的条件也可以放在 while 后面的表达式中，如例 4.8 所示。

【例 4.8】使用 while 语句编写一个程序，步骤如下。

打开记事本，编写程序代码如下，并保存为 WhileDemo2.java。

```
public class WhileDemo2{
    public static void main(String[] args){
        int i = 0;              //定义循环变量
        //输出 200 个”OK”
        while(i++<200){         //后缀自增运行符++，先将 i 与 200 比较，然后将 i 的值增加 1
            System.out.println("OK");
        }
    }
}
```

在上面这个程序中，对自变量 i 的值的改变，放在 while 后面的括号中。每次将 i 与 200 比较大小后，i 的值自动增加 1。如果在 while 语句中，没有改变循环条件的语句，那么循环将无限期地执行下去，这称为“死循环”。

【例 4.9】死循环的程序示例。

```
public class WhileDemo3{
    public static void main(String[] args){
        int i = 0;          //定义循环变量
        //输出 200 个”OK”
```

```
        while(i<200){    //在 while 循环中，没有改变循环变量 i 的条件，所以为死循环
            System.out.println("OK");
        }
    }
}
```

在上例中，因为没有改变循环变量 i 的值的语句，所以条件“i<200”永远成立，循环将会无限期地执行下去。这在编写程序时是要避免的。while 语句的流程图如图 4.5 所示。

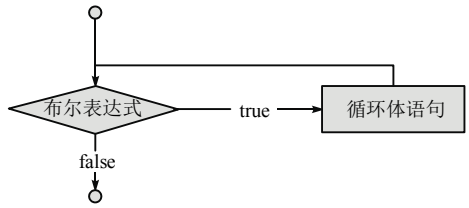


图 4.5　while 语句的流程图

【例 4.10】使用 if 语句和 while 语句编写一个程序，求 100（含 100）之内所有偶数的和，步骤如下。

① 打开记事本，编写程序代码如下，并保存为 EvenDemo.java。

```
// 求 100（含 100）之内所有偶数的和
public class EvenDemo{
    public static void main(String[] args){
        int number = 1, sum=0;          //定义两个整型变量
        while(number <= 100){           //当变量 number 的值不大于 100 时
            if(number%2==0)             //判断 number 是否为偶数
                sum = sum + number;     //将偶数的值加到变量 sum 上
            number++;                   //number 变量的值自增 1
        }
        System.out.println(“100(含 100)之内所有偶数的和是:” + sum);
    }
}
```

② 使用如下命令编译和运行 EvenDemo.java 源程序。

```
javac EvenDemo.java
java EvenDemo
```

③ 程序输出结果如下。

```
100(含 100)之内所有偶数的和是：2550
```

在程序 EvenDemo 中，从变量 number 值为 1 开始循环，直到条件 number<=100 不成立为止。在每一次循环中，都判断变量 number 能否被 2 整除。如果能被 2 整除，就将其值累加到变量 sum 上，最后循环结束，变量 sum 的值就为 100 以内偶数的和。

4.2.2　do-while 循环语句

与 while 语句功能相似的另一个循环语句是 do-while 语句。在大多数情况下，while 语句和 do-while 语句完成相同的功能，它们的作用是等价的。do-while 语句的基本语法格式如下。

```
do{
    语句或语句块
}while(布尔表达式);
```

说明：

- 表达式可以是而且必须是任何运算结果为布尔值的表达式。
- 如果只有一条语句的话，可以省略 do 后面的大括号。但建议不论几条语句，都使



用大括号括起来。

➢ while 后面一定要有分号（;），以示 do-while 语句结束，这与 while 语句不同。

do-while 循环语句的执行过程如下。

（1）首先执行 do 后面的循环体语句。

（2）然后对 while 后面小括号中的布尔表达式的值进行判断，如果为 true，再次执行 do 后面的循环体语句，并再次对布尔表达式的值进行判断；否则，结束循环语句。

（3）重复执行上述两步。

【例 4.11】使用 do-while 语句编写一程序，为一个数组赋初值，步骤如下。

```
public class DoWhileDemo1{
    public static void main(String[] args){
        int[ ] mathScore = new int[200];
        //为数组中的所有元素赋初值 60
        int i = 0;                              //定义循环变量
        do{                                     //do-while 循环语句
            mathScore[i] = 60;                  //为每一个数组元素赋初值 60
            i++;                                //循环变量 i 改变，自增 1
        }while(i<200);                          //判断 i<200 是否成立
    }
}
```

这与上一节使用的 while 语句一样，实现了相同的功能。改变循环的条件也可以放在 while 后面的表达式中，如下面的代码所示。

```
public class DoWhileDemo2 {
    public static void main(String[] args){
        int[ ] mathScore = new int[200];    //声明大小为 200 的一个整型数组
        //为数组中的所有元素赋初值 60
        int i = 0;                          //定义循环变量
        do{                                 //执行 do-while 循环语句
            mathScore[i] = 60;              //为每一个数组元素赋初值 60
        } while(i++ < 200);                 //将循环变量 i 放在条件表达式中
    }
}
```

如果在 do-while 语句中，没有改变循环条件的语句，那么循环将无限期地执行下去，这称为“死循环”，如下面的代码所示。

```
public class DoWhileDemo3{
    public static void main(String[] args){
        int i = 0;                          //定义循环变量
        //输出 200 个” OK”
        do{                         //执行 do-while 循环语句
            System.out.println("OK");
        } while( i< 200 );          //判断循环条件。如果条件成立，则继续执行 do 语句块
    }
}
```

在上例中，因为没有改变循环变量 i 的值的语句，所以条件 i<200 永远成立，循环将会无限期地执行下去。这在编写程序时是要避免的。do-while 语句的流程图如图 4.6 所示。

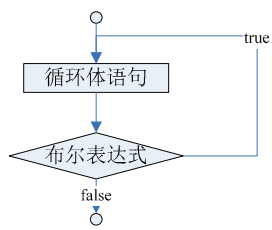


图 4.6　do-while 语句的流程图

【例 4.12】使用 if 语句和 do-while 语句编写一个程序，求 100（含 100）之内所有偶数的和。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 EvenDemo2.java。

```
// 求 100（含 100）之内所有偶数的和
public class EvenDemo2{
    public static void main(String[] args){
        int number = 1, sum=0;      //声明两个整型变量 number 和 sum，并赋初值
        do{
            if(number%2==0)         //判断 number 是否为偶数
                sum = sum + number; //如果是偶数，则加总各 sum
            number++;               //变量 number 自增
        } while(number <= 100);     //对变量 number 的大小进行判断。如果条件成立，则继续执行 do 语句块
        System.out.println(“100(含 100)之内所有偶数的和是:” + sum);
    }
}
```

② 使用如下命令编译和运行 EvenDemo2.java 源程序。

```
javac EvenDemo2.java
java EvenDemo2
```

③ 程序输出结果如下。

```
100(含 100)之内所有偶数的和是：2550
```

程序 EvenDemo2 与 EvenDemo 相似，也是求 100 以内偶数的和，但使用的是 do-while 语句。从结果上看，两个程序的运行结果都是一样的。

虽然 while 语句和 do-while 语句在大多数情况下可以相互替代，是等价的，但是在某些情况下，它们的使用还是有区别的。while 语句和 do-while 语句的区别如下。

- while 语句是先判断，后执行；而 do-while 语句是先执行，后判断。所以即使一开始循环条件就不成立，do-while 语句中的循环体也会执行一次。
- 一般情况下，使用 while 和 do-while 是等价的，同样的循环，使用 while 和使用 do-while 的结果是一样的。但是，如果在循环语句一开始布尔表达式就不成立（表达式的值为 false），那么它们的使用结果是不同的。例如下面两个示例：

```
public class Test1{
    public static void main(String[] args){
        boolean flag = false;           //声明布尔变量 flag，并赋初值 false
        int  value = 0;                 //声明整型变量 value，并赋初值 0
        while(flag){                    //如果 flag 的值为真，就进入循环体执行
            value++;
        }
        System.out.println("value=" + value);      //输出循环语句执行以后 value 的值
    }
}
```

在 Test1 中，使用了 while 循环语句，对布尔类型的条件变量 flag 的值进行判断。如果 flag 的值为真，就将变量 value 的值增 1。循环执行，直到 flag 的值为 false 为止，最后



输出执行循环语句以后的变量 value 的值。编译并运行 Test1，其输出结果如下。

```
value=0
```

下面的 Test2 与 Test1 几乎一样，除了使用了 do-while 循环语句，先执行 value 自增的操作，然后再对布尔变量 flag 的值进行判断，其代码如下所示。

```
public class Test2{
    public static void main(String[] args){
        boolean flag = false;           //声明布尔变量 flag，并赋初值 false
        int  value = 0;                 //声明整型变量 value，并赋初值 0
        do{                             //先执行一次循环体
            value++;                    //对变量 value 的值进行自增
        } while(flag);                  //后对布尔变量 flag 的值进行判断
        System.out.println("value=" + value);
    }
}
```

在 Test2 中，使用了 do-while 循环语句。不管布尔变量的初值是否为 false（即不论初始条件是否成立），都会先执行 value++语句。编译并运行 Test2，其输出结果如下。

```
value=1
```

4.2.3　for 循环语句

for 语句是最经常使用的循环语句，一般用在循环次数已知的情况下。在很多情况下，可以使用 for 语句替代 while 和 do-while 语句。其语法格式如下。

```
for(初始化表达式；条件表达式；迭代语句){
    循环体语句
}
```

说明：

- 初始化表达式通常用来对循环变量进行初始化，或者定义循环变量并初始化，它在循环过程中只会被执行一次。
- 条件表达式是一个布尔表达式，即运算结果为布尔值的表达式。只有当条件表达式为 true 时，才会执行或继续执行 for 语句的循环体语句。
- 迭代表达式用于改变循环条件的语句，为继续执行循环体语句做准备。这里也可以是一个运算表达式列表，包含一个或多个运算表达式。通常这里的表达式用来改变循环变量的值，如自增、自减等运算。

for 循环语句的执行过程如下。

（1）执行初始化表达式。

（2）对中间的条件表达式的值进行判断，如果为 true，执行后面的循环体语句。

（3）执行迭代表达式，改变循环变量的值。

（4）重复执行上述两步，开始下一次循环，直到某次中间的条件表达式的值为 false，结束整个循环语句。

【例 4.13】使用 for 语句示例。

```
public class ForDemo{
    public static void main(String[] args){
```

```
            int[ ] mathScore = new int[200];
            //为数组中的所有元素赋初值 60
            int i;                    //定义循环变量
            for(i=0;i<200;i++){       //循环语句，初始化 i 的值为 0；当 i<100 时，循环继
续；步长为 1
                mathScor[i] = 60;
            }
        }
    }
```

在上面这个示例中，程序执行到 for 语句时，先对循环变量 i 赋初值 0，然后判断条件表达式 i<200 的值，如果为 true，就执行循环体语句，为第 i+1 个元素赋值。执行完循环体语句，执行 i++，更新循环变量。然后再次对条件表达式 i<200 进行判断，开始下一次循环。

for 语句的使用相当灵活，循环变量既可以如上例中一样在 for 语句外面声明，然后在 for 语句中初始化，也可以全部放在 for 语句外面定义并初始化，或者全部放在 for 语句中定义并初始化。如下面的代码所示。

```
public class ForDemo {
    public static void main(String[] args){
        int[ ] mathScore = new int[200];          //声明大小为 200 的整型数组
        //为数组 ForDemo 中的所有元素赋初值 60
        int i = 0;                                //定义循环变量
        for(;i<200;i++){                          //循环 200 次
            mathScor[i] = 60;                     //为数组中的每个元素赋初值 60
        }
    }
}
```

如上例所示，如果在 for 语句之前声明并初始化循环变量，也是可以的。此时，可以省略 for 语句中的初始化表达式，但是分隔三个表达式的分号“;”不能省略，即使表达式为空。在 for 语句中，甚至三个表达式都为空也是合法的，但是分号依然不能省略，如以下代码所示。

```
for(;;){
…
}
```

对循环变量的声明和初始化也可以放在 for 语句中，毕竟如果初始化变量仅仅用于 for 循环的话，在 for 语句内声明更加合适。如下面的示例所示：

```
public class ForDemo {
    public static void main(String[] args){
        int[ ] mathScore = new int[200];    //声明大小为 200 的整型数组
        //为数组中的所有元素赋初值 60
        for(int i = 0;i<200;i++){           //循环 200 次
            mathScor[i] = 60;               //为数组中的每个元素赋初值 60
        }
    }
}
```

对于循环变量 i 的改变，也可以放在循环体内，但是在 for 语句后面的括号中，用于分隔表达式的分号不能省略，如下面的代码所示。

```
for(int i = 0;i<200;){                          //用于分隔表达式的分号不能省略
    mathScor[i] = 60;                           //为数组中的每个元素赋初值 60
    i++;                                        //改变循环变量的值
}
```

for 语句中的初始化表达式和迭代语句表达式，也可以是表达式列表，表达式之间用逗号分隔。如下面的代码所示。

```
for(int i = 0, j=0;i<200 && j < 100;i++,j--){
    …
}
```

for 语句的流程图如图 4.7 所示。

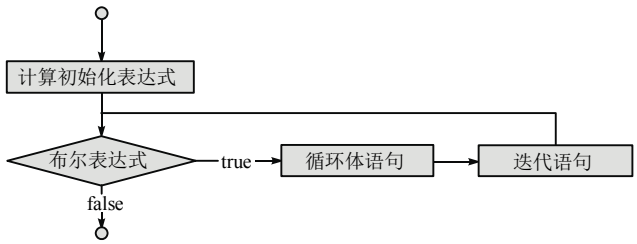


图 4.7　for 语句的流程图

【例 4.14】使用 for 语句编写一个程序，求 100（含 100）之内所有偶数的和。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 EvenDemo3.java。

```
// 求 100（含 100）之内所有偶数的和
public class EvenDemo3{
    public static void main(String[] args){
        int  sum=0;
        for(int number = 1;number <=100;number++){ //for 循环语句
            if(number%2==0)          //判断 number 是否为偶数
                sum = sum + number; //如果 number 是偶数，就累加到 sum 变量上
        }
        System.out.println("100(含 100)之内所有偶数的和是：" + sum);
    }
}
```

② 使用如下命令编译和运行 EvenDemo3.java 源程序。

```
javac EvenDemo3.java
java EvenDemo3
```

③ 程序输出结果如下。

```
100(含 100)之内所有偶数的和是：2550
```

可以看出，for 语句、while 语句和 do-while 语句通常情况下是等价的。到底在编程时使用哪一种循环语句，更多地取决于读者自己的兴趣。

4.2.4　循环语句的嵌套

一个循环结构的循环体中还可以包含另一个完整的循环结构，这称为循环的嵌套。嵌

套可以是多层的，既可以在内嵌的循环结构内，还可以再嵌套其他的循环结构。while 语句、do-while 语句及 for 语句都可以嵌套，而且它们之间也可以相互嵌套。下面是几种嵌套的语法形式。

（1）while 语句的嵌套。

```
while(条件表达式 1){
    while(条件表达式 2){
…
    }
}
```

（2）do-while 语句的嵌套。

```
do{
     do{
         …
    } while(条件表达式 1);
} while(条件表达式 2);
```

（3）for 语句的嵌套。

```
for(; ;){
    for(; ;){
    …
    }
}
```

（4）for 循环与 while 循环嵌套。

```
for(; ;){
    while(条件表达式){
    …
    }
}
```

（5）for 循环与 do-while 循环嵌套。

```
for(; ;){
    do{
        …
    } while(条件表达式);
}
```

（6）while 循环与 for 循环的嵌套。

```
while(条件表达式){
    for(; ;){
        …
    }
}
```

（7）do-while 循环与 for 循环的嵌套。

```
do{
    for(; ;){
        …
    }
} while(条件表达式 1);
```



（8）do-while 循环与 while 循环的嵌套。

```
do{
    while(条件表达式 1){
        …
    }
} while(条件表达式 2);
```

（9）while 循环与 do-while 循环的嵌套。

```
while(条件表达式 1){
    do{
        …
    } while(条件表达式 2);
}
```

【例 4.15】嵌套循环示例，使用循环的嵌套来处理二维数组。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 NestedForDemo.java。

```
public class NestedForDemo{
    public static void main(String[] args){
        int  i , j;                                  //声明两个循环变量
        int[ ][ ] arrays = new int[5][5];            //定义一个二维数组
        //为二维数组赋值
        for(i=0;i<5;i++){
            for(j=0;j<5;j++){
                arrays[i][j] = i + j ;  //第 i 行第 j 列的元素的值为 i 和 j 的和
            }
        }
        //输出二维数组中的元素值
        for(i=0;i<5;i++){                    //对行进行循环
            for(j=0;j<5;j++){                //对每一行中的列进行循环
                System.out.print (arrays[i][j] + "  ");
                                             //输出二维数组第 i 行第 j 列元素的值
            }
            System.out.println("");           //换行
        }
    }
}
```

② 使用如下的命令编译和运行 NestedForDemo.java 源程序。

```
javac NestedForDemo.java
java NestedForDemo
```

③ 程序运行结果如图 4.8 所示。

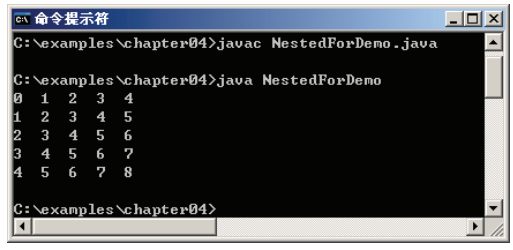


图 4.8　二维数组输出结果（循环的嵌套）

在程序 NestedForDemo 中，通过嵌套的双重循环 for 语句，改变二维数组下标 i 和 j 的值，依次为二维数组的元素赋值及输出二维数组中每个元素的值。

4.2.5　使用增强型 for 语句遍历一维数组中的元素

对于简单的循环，Java 提供了一种 foreach 语法，又称为“增强型的 for 循环语句”，

其应用的对象之一是在数组的依次存取上。在某些情况下，使用 foreach 更直观、更简洁。foreach 语法如下所示。

```
for(数据类型  数据变量：数组){
    使用数据变量的循环体语句组
}
```

因此，对于数组元素的遍历，如果使用上一节中讲到的 for 语句来遍历一维数组中的元素的话，通常代码如下。

```
int[ ] arrays = {1,2,3,4,5};              //声明并初始化整型数组
for(int i=0;i<arrays.length;i++){         //使用 for 循环语句遍历数组元素
    System.out.pritnln(arrays[i]);
}
```



上面用到了一个数组对象的属性：arrays.length。这在“数组”一章没有提到。在 Java 语言中，是将数组看成对象的，对象有一个属性 length（属性的概念在“类和对象”一章中会讲到），代表数组的大小，这样程序员无须知道数组的长度就可以遍历数组。

如果使用 foreach 语法，可以将上述代码改写为：

```
int[ ] arrays = {1,2,3,4,5};              //声明并初始化整型数组
for(int element : arrays){                //使用 foreach 循环语句遍历数组元素
    System.out.pritnln(element);
}
```

第一次循环中，从数组 arrays 中依次取出的元素，会自动赋给变量 element，程序员不用自行判断是否超出了数组的长度。注意，element 的类型必须与数组元素的数据类型相同。增强型的 for 语句也可用于数组元素是对象的情形，请看下面的代码：

```
String[ ] names = {"张小明","刘芳","欧阳光明"};   //声明并初始化字符串类型数组
for(String name : names){                 //使用 foreach 循环语句遍历数组元素
    System.out.pritnln(name);
}
```

4.2.6 对一维数组进行排序

在编程的时候，经常需要对数组中的数据进行排序处理，例如价格从高到低排列、年龄从大到小排列等，这需要使用到循环语句。请看下面对一维数组进行排序的示例。

【例 4.16】使用“冒泡排序法”对一维数组进行排序。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 SortDemo.java：

```
public class SortDemo{
    public static void main(String[] args){
        int  i , j;                       //定义循环变量 i 和 j
        int[ ] arrays = {83，26，5，18，72};
        int length = arrays.length;       //获取数组的长度，并保存在变量 length 中
        System.out.println("排序前，数组元素序列是：");
        for(i=0;i<length;i++){            //遍历数组当中的每一个元素，并输出
            System.out.print(arrays[i] + "  ");
        }
```

```
        System.out.println("");
        //使用双重循环对数组排序
        for(i=0;i<length-1;i++){                    //循环比较 length-1 趟
            for(j=0;j<length-1-i;j++){              //对每一趟进行相邻元素的比较
                int temp=0;
                if(arrays[j]>arrays[j+1]){          //相邻两个元素进行互换
                    temp = arrays[j];
                    arrays[j] = arrays[j+1];
                    arrays[j+1] = temp;
                }
            }
        }
        //输出排序后的数组元素序列
        System.out.println("排序以后，数组元素序列是：");
        for(i=0;i<length;i++){                      //循环遍历输出数组元素的值
            System.out.print(arrays[i] + "  ");
        }
    }
}
```

② 使用如下的命令编译和运行 SortDemo. java 源程序。

```
javac SortDemo.java
java SortDemo
```

③ 程序运行结果如图 4.9 所示。

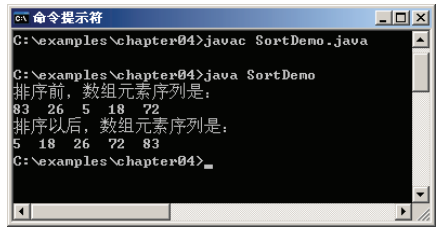


图 4.9　一维数组排序

在这个程序中，用冒泡排序法对一维数组进行了排序。所谓冒泡排序法，就是将相邻的元素依次进行比较，如果前面的元素比后面的元素值大，就交换它们的值。这样每一轮排序下来，最大的元素会被交换到数组的最后一个位置。这样一轮一轮比较和交换，就像冒泡一样。

4.2.7　使用增强型 for 语句遍历二维数组中的元素

通常情况下，遍历二维数组使用双重 for 循环。但是也可以使用增强型的 for 语句来遍历二维数组。使用前面讲到的 for 语句来遍历二维数组中的元素，通常代码如下。

```
int i , j;
int[ ][ ] arrays = {{1,2,3},{4,5,6},{7,8,9}};
for(i=0;i<3;i++){                                   //对行进行循环
    for(j=0;j<3;j++){                               //对每一行中的列进行循环
        System.out.pritnln(arrays[i][j]);
    }
}
```

前面讲二维数组时讲过，二维数组实际上是数组的数组，就是说，可以把二维数组看成是这样的一维数组：它的元素又是一维数组。这样的话，就理解如何使用 foreach 语法进行存取了。如果使用 foreach 语法，可以将上述代码改写为：

```
int i , j;
int[ ][ ] arrays = {{1,2,3},{4,5,6},{7,8,9}};
```

```
for(int[]  row : arrays){        //循环遍历数组 arrays 的元素（同样是一维数组）
    for(int element : row){      //循环元素数组中的每一个元素
        System.out.pritnln(element);
    }
}
```

第一次循环中，从数组 arrays 中依次取出的元素，会自动赋给变量 element，程序员不用自行判断是否超出了数组的长度。注意，element 的类型必须与数组元素的数据类型相同。

增强型的 for 语句也可用于数组元素是对象的情形，请看下面的代码：

```
String[ ] names = {"张小明","刘芳","欧阳光明"};
for(String name : names){
    System.out.pritnln(name);
}
```

4.3 跳转语句

在前面的循环结构中，当循环条件不满足时或循环次数达到要求时，循环会退出。但是，有时候可能需要在循环的执行过程中，当发生了某种情况之后，提前终止循环，这就需要跳转语句 break 和 continue。

4.3.1 break 跳转语句

在 switch 语句中，已经接触到了 break 语句。在 switch 语句中，break 语句的作用是终止 switch 语句的执行，而整个程序继续执行后面的语句。用在循环结构中的 break 语句，也起同样的作用。

当在循环结构中执行到 break 语句时，它立即停止当前循环的执行，即提前结束循环，接着执行循环下面的语句。break 语句的语法格式如下。

```
break;
```

与在 switch 中一样，直接在 break 后面加上分号“;”就可以了。如下面的代码片段所示。

```
final int PI = 3.14;             //定义一个常量 PI，代表圆周率
for(r=1;r<=10;r++){
    area = PI * r * r ;          //求圆的面积，r 为半径
    if(area>100)    break;
    System.out.println(area);
}
```

这个代码片段计算 r=1 到 r=10 时的圆面积，直到圆的面积 area 大于 100 为止。从上面的 for 循环可以看到：当 area>100 时，执行 break 语句，提前结束循环，即不再继续执行其余的几次循环。break 语句不能用于循环语句和 switch 语句之外的任何其他语句中。

【例 4.17】编写程序，功能为在数组中查找是否有指定的整数，如果有，指出其所在位置。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 BreakDemo.java。

```
public class BreakDemo{
    public static void main(String[] args){
        int  i;                          //指定变量出现的位置
        int  number = 45;                //定义要查找的变量
```



```
            //int number = 110;
            int[ ] arrays = {23,45,2,8,67,258,94,1,18,45,1250};
            for(i=0;i<arrays.length;i++){   //遍历数组元素
                if(arrays[i] == number){    //如果第 i 个元素的值与 number 变更的值
相等
                    break;                  //中止循环，跳出 for 语句
                }
            }
            if(i < arrays.length){          //如果 for 语句是因 break 终止，说明有符
合要求的数据
                System.out.println("您要查找的" + number + "是数组的第" + (i+1) +
"个元素");
            }else{
                System.out.println("对不起，没有您要找的数据！");
            }
        }
    }
```

② 使用如下命令编译和运行 SortDemo.java 源程序。

```
javac BreakDemo.java
java BreakDemo
```

③ 程序运行后输出结果如下。

```
"您要查找的 45 是数组的第二个元素"
```

如果将变量 number 的值改为 110，那么输出的结果如下。

```
"对不起，没有您要找的数据！"
```

如果有嵌套的循环，而 break 语句在内层循环中被执行，那么程序将跳出内层循环，从下一个外层循环开始执行。

【例 4.18】嵌套循环中 break 语句应用示例。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 BreakDemo2.java。

```
public class BreakDemo2{
    public static void main(String[] args){
        for(int i=0;i<3;i++){
            System.out.println("Pass " + i + ": ");
            for(int j=0;j<100;j++){
                if(j==10)  break;           //如果变量 j 等于 10，终止内层循环
                System.out.print(j + "  ");
            }
            System.out.println( );          //换行
        }
        System.out.println("循环结束. ");
    }
}
```

② 使用如下命令编译和运行 BreakDemo2.java 源程序。

```
javac BreakDemo2.java
java BreakDemo2
```

③ 程序运行后输出结果如下。

```
Pass 0: 0  1  2  3  4  5  6  7  8  9
Pass 1: 0  1  2  3  4  5  6  7  8  9
```

```
Pass 2: 0  1  2  3  4  5  6  7  8  9
循环结束
```

从中可以看出，在内部循环中的 break 语句仅仅终止了该循环，外部的循环不受影响。使用 break 需要注意以下几点：

- 一个循环中可以有一个以上的 break 语句，但是太多的 break 语句会破坏代码结构。
- switch 语句中的 break 仅仅影响该 switch 语句，而不会影响其中的任何循环。
- break 不是为提供一种正常的循环终止的方法而设计的。循环的条件语句是专门用来终止循环的。只有在某类特殊的情况下，才用 break 语句来取消一个循环。

从上面的示例中可以看出，在内层的 break 语句只是终止它所在的循环。如果当某种特殊情况发生时，想同时终止所有的内外层循环，即直接终止所有的循环，那么需要在每一层中都使用 break 语句。例如下面的代码片段所示。

```
for(int i=0;i<3;i++){
boolean flag = false;
for(int j=0;j<100;j++){
    if(j==10){
        break;                  //如果变量 j 等于 10，终止内层循环
        flag = true;
    }
    if(flag)  break;            //如果终止了内层循环，那么外层循环也终止
}
}
```

上述代码并无太大的实际意义，它仅仅展示了如果要同时跳出两层嵌套的循环时，该如何做。看上去还是比较麻烦的，设想一下，如果嵌套的不是两层，而是五层，那么代码如何写？显然，更麻烦！很幸运，Java 语言提供了一种扩展的 break 语句，称为标签 break。标签 break 可以终止一个或几个代码块，其语法格式如下。

```
break 标签名称;
```

这里，标签名称是标识代码块的标签。当这种形式的 break 执行时，控制被传递出指定的代码块。被加标签的代码块必须包围 break 语句。请看下面的示例，注意其中的注释处的标签。

【例 4.19】标签 break 语句应用示例。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 BreakDemo3.java：

```
public class BreakDemo3{
    public static void main(String[] args){
        outer:                                  //定义标签，标签名称为 outer
        for(int i=0;i<3;i++){
            System.out.print("Pass " + i + ": ");
            for(int j=0;j<100;j++){
                if(j==10)  break outer;         //如果变量 j 等于 10，终止所有循环
                System.out.print(j + "  ");
            }
            System.out.println("这个语句不会被执行");      //换行
        }
        System.out.println("循环结束. ");
    }
}
```



在上面的代码中，定义了一个标签 outer。标签可以是任何合法有效的 Java 标识符，后跟一个冒号。一旦给一个块加上标签以后，就可以使用这个标签作为 break 语句的对象了。

② 使用如下命令编译和运行 BreakDemo3.java 源程序。

```
javac BreakDemo3.java
java BreakDemo3
```

③ 程序运行后输出结果如下。

```
Pass 0: 0  1  2  3  4  5  6  7  8  9  循环结束
```

可以看到，当内部循环退到外部循环时，两个循环都被终止了。标签 break 还可以用于任何的代码块或嵌套代码块，它会使程序的执行在加标签的块的结尾处重新开始。例如，下面的程序示例了三个嵌套块，每一个都有它自己的标签。其中的标签 break 语句使执行流程向前跳，跳过了两个 println()语句。

【例 4.20】嵌套的标签 break 语句应用示例。编写程序的步骤如下。

① 打开记事本，编写程序代码如下，并保存为 BreakDemo4.java。

```
public class BreakDemo4{
    public static void main(String[] args){
        boolean flag = true;
        first:                          //定义标签，标签名称为 outer
        {
            second:                     //定义嵌套的标签，标签名称为 second
            {
                third:                  //定义第三层嵌套的标签，标签名称为 third
                {
                    System.out.println("这里是第三层，执行 break 之前. ");
                    if(flag)
                        break second;           //跳转到第二层循环结束处
                    System.out.println("这条语句不会被执行");
                }
                System.out.println("这里是第二层，这一句也不会被执行。");
            }
            System.out.println("这里是第一层，这一句会被执行。");
        }
    }
}
```

② 使用如下命令编译和运行 BreakDemo4.java 源程序。

```
javac BreakDemo4.java
java BreakDemo4
```

③ 程序运行后输出结果如图 4.10 所示。

在程序 BreakDemo4 中，当条件 flag 为 true 时，就会执行“break second”语句，跳转到标签 second 所代表的语句块外，因此该语句及 second 标签所标识的语句块后面的语句都不会被执行。

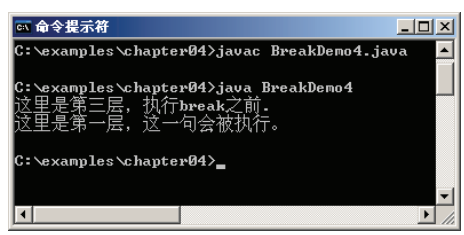


图 4.10　标签 break 语句的使用

4.3.2　continue 跳转语句

如果需要提前结束本次循环，立即开始下一轮循环，而不是终止整个循环的执行，可

以使用 Java 提供的 continue 语句。continue 语句的语法形式如下。

```
continue;
```

continue 应用在 for、while、do-while 等循环语句中，其作用为结束本次循环，即跳过循环体中下面尚未执行的语句，接着进行下一次是否执行循环的判断。

【例 4.21】continue 跳转语句应用示例。

```
//把 100～200 之间的不能被 3 整除的数输出
public class ContinueDemo{
    public static void main(String[] args){
        int  n;                                  //声明变量 n
        for(n=100;n<=200;n++){                   //从 100 循环到 200
            if(n%3 ==0)                          //如果整数 n 能被 3 整除
                continue;                        //则结束本次循环，开始下一循环
            System.out.print (n + " ");          //如果不能被 3 整除，则输出 n 的值
        }
    }
}
```

在上面的程序中，当整数 n 能被 3 整除时，执行 continue 语句，结束本次循环（即跳过后面的输出语句）。只有当 n 不能被 3 整除时才执行后面的输出函数。

continue 和 break 语句都是跳出循环，但它们的作用是不同的。continue 语句只结束本次循环，而不是终止整个循环的执行。而 break 语句则是结束整个循环过程，不再判断执行循环的条件是否成立。如果有以下两个循环结构：

- 循环结构 1

```
while(表达式 1){
    …
    if(表达式 2)  break;
    …
}
```

- 循环结构 2

```
while(表达式 1){
    …
    if(表达式 2)  continue;
    …
}
```

那么，循环结构 1 的流程图如图 4.11 所示，而循环结构 2 的流程图如图 4.12 所示。请注意两图中当“表达式 2”为 true 时流程的转向。

4.4 实例

循环结构既有 while、do-while 语句，也有 for 和 foreach 语句，使用起来非常灵活。而且，在很多情况下，这几种循环结构是可以相互替代的。本节就通过译密码和打印九九乘法表两个实例程序的讲解，帮助读者学习和掌握循环结构在程序中的应用。

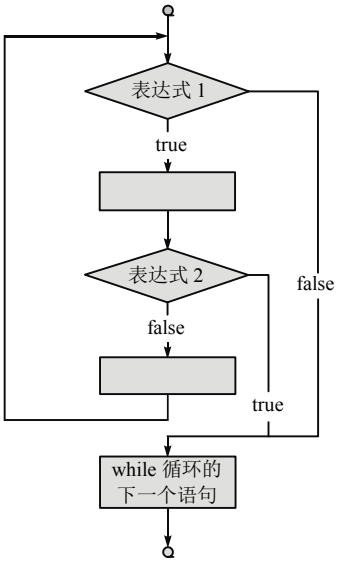


图 4.11　使用 break 语句的流程图

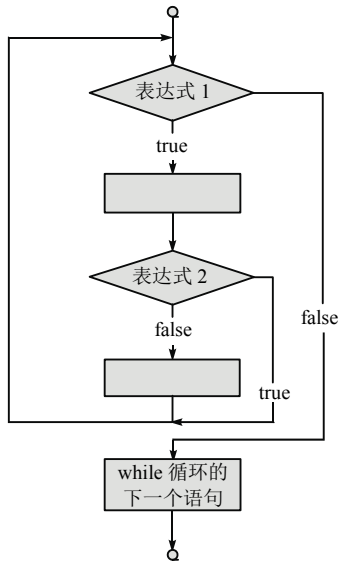


图 4.12　使用 continue 语句的流程图

4.4.1　实例 1：译密码

所谓译密码，是指为了通信保密，按一定规律，将通信内容转换成密码，收到加密内容的收信人再按约定的规律将其译回原文。本实例中，将按以下规律将通信内容转换成密码。

将字母 A 变成字母 E，a 变成 e，即将原文中的字母变成其后的第 4 个字母。而 W 变成 A，X 变成 B，Y 变成 C，Z 变成 D，即循环加密。字母按上述规律转换，非字母字符不变。例如“love”变成“pszi”。下面是加密程序的代码。

```
    //通信内容简单加密
    public class Encrypt{
        public static void main(String[] args){
            char[ ] message = {'I', ' ', 'l', 'o', 'v', 'e', ' ', 'y', 'o', 'u',
'! '};  //定义要加密的字符数组
            System.out.println("加密之前，原文内容为：");  //先输出加密之前的内容
            for(int i=0;i<message.length;i++){          //遍历字符数组
                System.out.print(message[i]);           //输出数组中每一个字符
            }
            System.out.println( );
            System.out.println("加密之后，密文内容为：");
            for(int i=0;i<message.length;i++){      //遍历字符数组的长度
                int character = message[i];         //获取字符串中第 i+1 个字符
                //如果为字符，则进行转换
                if((character >='a' && character<='z') || (character >='A' &&
character<='Z')){
                    character = character + 4;      //每个字符的 ASCII 码值加 4
                    //如果换算以后的字符超出了字母表的范围，则从头再循环计算
                    if((character>'Z' && character<='Z'+4) || character>'z')
                        character = character - 26;
                }
                System.out.print((char)character);
```

```
            }
        }
    }
```

在程序中，对原始字符的处理办法是：先判定它是否是大写字母或小写字母，若是，则将其值加 4（变成其后的第 4 个字母）。如果加 4 以后字符值大于‘Z’或‘z’，则表示原来的字母在 V（或 v）之后，就将它转换为 A～D（或 a～d）之一。办法是使其减 26（请查阅 Java 的字符编码表 Unicode 表）。注意在最后输出时，要使用强制类型转换，将整型转换为字符型输出。输出结果如图 4.13 所示。



图 4.13　加密结果

按照同样的规律，可以写出相应的解码程序，如下所示。

```
//通信内容解密
public class Decode{
    public static void main(String[] args){
        char[ ] message = {'M',' ','p','s','z','i',' ','c','s','y', '! '};
        System.out.println("解密之前，密文内容为：");
        for(int i=0;i<message.length;i++){ //循环输出字符数组中的每一个字符
            System.out.print(message[i]);
        }
        System.out.println( );
        System.out.println("解密之后，原文内容为：");
        for(int i=0;i<message.length;i++){ //遍历字符数组中的每一个字符
            int character = message[i];     //获取字符串中第 i+1 个字符
            //判断字符所属类型
            if((character >='a' && character<='z') || (character >='A' &&
character<='Z')){
                character = character - 4;
                if((character<='a' && character>='a'-4) || character<'A')
                    character = character + 26;
            }

System.out.print((char)character);
        }
    }
}
```

输出结果为如图 4.14 所示。

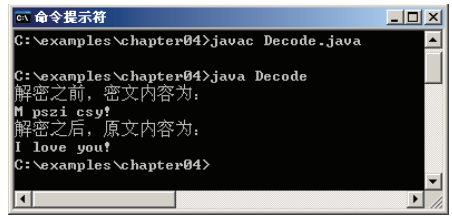


图 4.14　解密结果

4.4.2　实例 2：打印九九乘法表

九九乘法表是每个人小时候都要熟读并背诵的。九九乘法表既有行又有列，因此下面这个程序，使用双重循环来实现打印输出一个九九乘法表。

```
//打印九九乘法表
public class MultiTable{
    public static void main(String[] args){
        for(int i=1;i<10;i++){                    //循环行
            for(int j=1;j<=i;j++){                //循环列
```



```
                System.out.print(j + "*" + i + "=" + i*j);
                if(i*j<10)                          //如果乘法之和为个位数
                    System.out.print("   ");       //多输出一个空格，使输出结果排
列整齐
                else
                    System.out.print("  ");        //少输出一个空格
            }
            System.out.println( );
        }
    }
}
```

编译并运行 MultiTable.java，输出结果如图 4.15 所示。

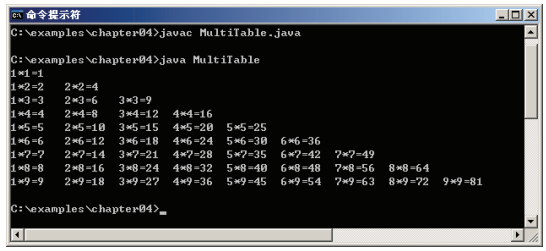


图 4.15　九九乘法表

在程序 MultiTable 中，注意嵌套的 for 循环的循环条件。因为这里要输出一个三角形的九九乘法表，所以第二层 for 循环的循环条件为 j<=i，即列索引值要小于等于行索引值，只有这样，才能在行和列索引值相等时就换行。另外，为了输出结果的美观整齐，当乘积为个位数时，后面要多输出一个空格。



4.5　小结

本章深入介绍了有关程序流程控制的语句，主要包括选择语句、循环语句和跳转语句。其中的 while 循环语句因其灵活方便，读者应重点掌握。本章最后讲到的跳转语句主要用于提高循环语句的灵活性，特别是标签 break 语句，是本章掌握的难点。建议读者在学习本章时，一定要将书中的实例调试和运行通过，以理解程序流程控制语句。

在第 5 章中，将向读者介绍 Java 编程中经常要遇到的问题：数字、字符和字符串的处理。

第5章 数字、字符和字符串处理

在程序中，最经常处理的数据就是数字、字符和字符串，因此，在各种程序设计语言中，都提供有专门用来处理数字、字符和字符串的功能。Java 语言也不例外，它提供有专门用来处理数字的 Number 及其子类，专门用来处理数学运算的 Math 类，专门用来处理字符的 Character 类，以及专门用来创建和操作字符串的类 String。这些类都位于 java.lang 包中。本章主要讲解这些类的用法。



5.1 数字

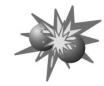
本节介绍 java.lang 包中的 Number 类及其子类，以及在什么条件下使用这些类的实例而不是使用原始数据类型。本节还会介绍 PrintStream 和 DecimalFormat 类，这两个类提供了输出格式化数字的方法。在本节的最后将介绍 java.lang 包中的 Math 类。Math 类包含有 Java 语言的数学运算函数，包括三角函数和指数函数等。

5.1.1 Numbers 数字包装器类

一般在写程序时，当使用到数字时，多数都是在代码中使用原始数据类型，如 int 类型、float 类型、byte 类型等。这样书写代码比较简单易懂。例如在下面的代码中，有三个不同的数据类型，分别表示三个数字。

```
int i = 320;
float price = 3.65f;
byte mask = 0xaf;
boolean flag = false;
```

然而，在面向对象的程序设计中，很多时候由使用对象来代替原始类型显得更加合理。从 JDK 5.0 开始，Java 平台为每一个原始数据类型都提供有“包装器（wrapper）”类。这些类将原始数据“包装”成对象。所以经常地，当程序中一个期望是对象的地方使用了原始数据，那么编译器会将原始数据“装箱”至其包装器中。同样地，当程序中一个期望是原始数据的地方使用了一个数字对象，编译器会将对象“拆箱”。这就是 JDK 5.0 中的自动装/拆箱新特性。下面是一个装箱和拆箱的示例。



```
Integer x,y;
x = 17;
y = 25;
System.out.println(x+y);
```

当 x 和 y 分别被赋予整型值 17 和 25 时，编译器会对这些整型值进行装箱操作，因为 x 和 y 都是整型对象。而在 println()语句中，x 和 y 对象被拆箱，这样它们才能被作为整数相加。所有的数字包装器类都是抽象类 Number 的子类，如图 5.1 所示。

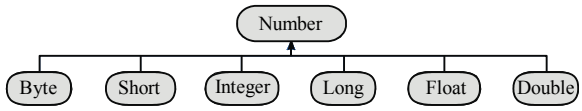


图 5.1　包装器类

对 Java 基本类型的自动装/拆箱使得代码更加简洁易懂。下面的这个例子示范了将一个 int 类型的数据存储到一个 ArrayList 动态数组中，并且再取出来。自动装/拆箱使得在 int 和 Integer 对象之间的转换能自动进行。

未使用自动装/拆箱以前的代码如下所示。

```
    ArrayList<Integer> list = new ArrayList<Integer>( );        //创建动态数组对象 list
    list.add(0,new Integer(17));              //构建数字 17 的 Integer 对象，并添加到数组中
    int sum = (list.get(0)).intValue( );    //从数组中获取 Integer 对象，并将其转换为数字
```

如果将上面的代码使用自动装/拆箱代码进行改写，那么代码如下所示。

```
    ArrayList<Integer> list = new ArrayList<Integer>( );   //创建动态数组对象 list
    list.add(0,17);                 //将数字 17 自动装箱，并添加到数组中
    int sum = list.get(0);          //从数组中获取 Integer 对象，自动拆箱为数字并赋值
```

从上面的对比中可以看出，使用自动装/拆箱的代码显得更加简洁易懂，符合程序员的思维习惯，并相应地减少了代码量。使用 Number 对象除了这个好处之外，还有三个理由支持使用 Number 对象而不是使用原始数据类型。

- 作为一个方法的参数对象（当处理数字集合时经常使用）。
- 方便地使用由 Number 类定义的常量，如 MIN_VALUE 和 MAX_VALUE，它们提供了相应数据类型的最大值和最小值。
- 方便地使用 Number 类的类方法用于将值转换到其他原始类型或从其他原始类型转换、转换到字符串或从字符串转换，以及在数字系统间进行转换（十进制、八进制、十六进制、二进制）。

表 5.1 列出了所有 Number 类的子类所实现的实例方法。

表 5.1　所有 Number 类的子类所实现的方法

方　法	描　述
byte byteValue() short shortValue() int intValue()	将当前 Number 对象的值转换为原始数据类型返回

续表

方　法	描　述
long longValue() float floatValue() double doubleValue()	将当前 Number 对象的值转换为原始数据类型返回
int compareTo(Byte anotherByte) int compareTo(Double anotherDouble) int compareTo(Float anotherFloat) int compareTo(Integer anotherInteger) int compareTo(Long anotherLong) int compareTo(Short anotherShort)	将当前 Number 对象与参数进行比较
boolean equals(Object obj)	判断当前数字对象是否与参数相等 如果参数不是 null，并且是一个与当前类型相同类型的对象，并且具有相同的数字值 对于 Double 和 Float 对象，在 Java API 文档中描述的有一些额外的要求

每一个 Number 类还包含有其他方法，这些方法用于转换数字到字符串及从字符串转换到数字，以及在数字系统之间进行转换。表 5.2 列出了 Integer 类中的这些方法。其他 Number 子类的这些方法与此相似。

表 5.2　Integer 类的转换方法

方　法	描　述
static Integer decode(String s)	将一个字符串解码到一个整数。能接收十进制、八进制、或十六进制数字的字符串表示作为输入
static int parseInt(String s)	返回一个整数（只能是十进制）
static int parseInt(String s,int radix)	返回一个整数，给定一个十进制、二进制、八进制、或十六进制（radix 分别等于 10,2,8 或 16）数字的字符串表示作为输入
String toString()	返回一个代表当前 Integer 值的 String 对象
Static String toString(int i)	返回一个代表指定整数的 String 对象
Static Integer valueOf(int i)	返回一个持有指定原始数据值的 Integer 对象
static Integer valueOf(String s)	返回一个持有指定字符串表示的值的 Integer 对象
static Integer valueOf(String s,int radix)	返回一个持有指定字符串表示的值的 Integer 对象，使用 radix 的值进行解析。例如，如果 s=”333”并且 radix=8，那么该方法返回与八进制数字 333 等价的基数为 10 的整数

5.1.2　数字的格式化输出

在此之前，读者可能已经注意到使用 print()和 println()方法用于打印字符串到标准输出（System.out）。因为所有的数字都能被转换到字符串，所以程序员可以使用这些方法来打印出任意一个字符串和数字的混合。不过，Java 程序设计语言提供有另外一些方法，当输出内容包含有数字时，程序员可以使用这些方法对打印输出进行更多的控制。

1．printf()和 format()方法

在 java.io 包中，包括有一个 PrintStream 类，这个类有两个格式化方法可以被用来替代 print()和 println()方法。这两个格式化方法是 format()和 printf()，它们彼此是等价的。大家已经很熟悉并且经常使用的 System.out 恰好就是一个 PrintStream 对象，因此可以在 System.out 上调用 PrintStream()方法。如此一来，程序员就可以在代码中先前任何使用 print()



或 println()方法的地方使用 format()或 printf()了。例如：

```
System.out.format(……);
```

这两个 java.io.PrintStream 方法的语法是相同的，如下所示。

```
public PrintStream format(String format,Object… args)
```

这里，参数 format 是一个字符串，指明了被使用的格式，参数 args 是要使用指定格式被打印的变量列表。请看下面的示例代码。

```
System.out.format("float 类型变量的值为%f，整型变量的值为%d，字符串类型为%s",floatVar,intVar,stringVar);
```

在这个代码中，第一个参数 format 是一个格式字符串，详细说明在第二个参数 args 中的对象是如何被格式化的。格式字符串包含纯文本，如“格式说明”，这是特殊的字符，用来格式化 Object… args 参数。



标注 Object… args 被称为“可变参数”，这意味着参数的个数是可变的。

格式说明以一个百分号“%”开始，以一个“转义符”结尾。转义符是一个字符，用来标示参数被格式化的类型。在百分号“%”和转义符之间，有可选的标志和说明。在 java.util.Formatter 文档中有许多转义符、标志和说明。下面是一个最基本的示例。

```
int i = 1012;
System.out.format("变量 i 的值为: %d%n", i);
```

在这个示例中，%d 说明有一个变量是一个十进制的整数。%n 是一个平台独立的换行符。这两行代码的输出结果如下。

```
变量 i 的值为:461012
```

方法 printf()和 format()是被重载的方法。它们每一个都有一个如下语法所示的版本。

```
public PrintStream format(Locale l, String format, Object... args)
```

例如，要在法语系统中打印输出数字（在法语系统中，表示浮点数时使用逗号代替英语表示中的圆点小数位），将使用如下所示的代码：

```
System.out.format(Locale.FRANCE, "float 类型变量的值为%f，整型变量的值为%d，"+"字符串类型为%s", floatVar,intVar,stringVar);
```

2．格式化数字输出应用示例

除了上面示例代码中所见的%d、%n 外，Java 中还提供了其他许多的转义符和格式标志。表 5.3 列出了 Java 语言中的一些常见的转义符和标志。

表 5.3　Java 中常见的转义符和标志

转 义 符	标　志	解释说明
d		一个十进制整数
f		一个浮点数
n		与运行应用程序的平台相适应的一个新的行的字符。应该总是使用%n，而不是\n
tB		一个日期和时间变换——月份显示为本地格式的全名称
td,te		一个日期和时间变换——月份中的天数显示为 2 位数字。td 根据需要会以 0 开头，而 te 不可以

续表

转 义 符	标　志	解释说明
ty,tY		一个日期和时间变换——ty=2 位数字的年，tY=4 位数字的年
tl		一个日期和时间变换——12 小时制中的小时
tM		一个日期和时间变换——分钟显示为 2 位数字，必要时以 0 开头
tp		一个日期和时间变换——显示为本地格式的 am/pm(小写)
tm		一个日期和时间变换——月份显示为 2 位数字，必要时以 0 开头
tD		一个日期和时间变换——日期等价于%tm%td%ty
	08	8 个字符宽，必要时以 0 开头
	+	包含符号，与正负无关
	,	包含本地格式的千分位字符
	-	左对齐（Left-justified.）
	.3	小数点后保留 3 位
	10.3	10 个字符宽，右对齐（right justified），小数点后保留 3 位

【例 5.1】下面的程序 TestFormat.java 展示了可以使用 format()进行的一些格式化。输出结果在代码嵌入的注释中用双引号显示。

```
import java.util.Calendar;
import java.util.Locale;
public class TestFormat {
    public static void main(String[] args) {
      long n = 382112;
      System.out.format("%d%n", n);                  //  输出结果为"382112"
      System.out.format("%08d%n", n);                //  输出结果为"00382112"
      System.out.format("%+8d%n", n);                //  输出结果为" +382112"
      System.out.format("%,8d%n", n);                //  输出结果为" 382,112"
      System.out.format("%+,8d%n%n", n);             //  输出结果为"+382,112"
      double pi = Math.PI;                           //  获取圆周率的值
      System.out.format("%f%n", pi);                 //  输出结果为"3.141593"
      System.out.format("%.3f%n", pi);               //  输出结果为"3.142"
      System.out.format("%10.3f%n", pi);             //  输出结果为"     3.142"
      System.out.format("%-10.3f%n", pi);            //  输出结果为"3.142"
      System.out.format(Locale.FRANCE,               //  用法语表示
                        "%-10.4f%n%n", pi);          //  输出结果为"3,1416"
      Calendar c = Calendar.getInstance();           //  获取表示当前日期的对象
      System.out.format("%tB %te, %tY%n", c, c, c); //  输出结果为"二月 12, 2009"
      System.out.format("%tl:%tM %tp%n", c, c, c);   //  输出结果为"5:52 上午"
      System.out.format("%tD%n", c);                 //  输出结果为"02/12/09"
    }
}
```

编译并运行上述程序代码，输出结果如下。

```
382112
00382112
 +382112
 382,112
+382,112

3.141593
3.142
```



```
3.142
3.142
3,1416

二月 12, 2009
5:52 上午
02/12/09
```

在上面的程序代码中，用到了两个类 Math 和 Calendar。其中 Math 类有很多静态方法，用来进行各种数学函数运算，本章稍后会讲到。Calendar 是封装了日期的类，调用其 getInstance()方法可以获取封装了当前日期的 Calendar 对象实例。

3．DecimalFormat 类

如果想要更加精确地控制输出数字的格式，如要控制数字开头和结尾的 0、前缀后缀、千分位分隔符、十进制分隔符等的显示，可以使用 java.text.DecimalFormat 类。DecimalFormat 类在格式化数字方面有很大的灵活性，不过如果不能正确使用，也可能会使代码更加复杂。请看下面的示例程序 DecimalFormatDemo.java 的代码，它详细说明了 DecimalFormat 类的使用。

【例 5.2】在下面的示例程序 DecimalFormatDemo.java 中，通过传递一个模式匹配字符串到 DecimalFormat 的构造器，创建了一个 DecimalFormat 对象 myFormatter。然后由 myFormatter 调用 format()方法，format()方法是由 DecimalFormat 继承自 NumberFormat 类的。format()方法接收一个 double 类型的值作为参数，并以字符串形式返回格式化后的数字。

```
import java.text.*;
public class DecimalFormatDemo {
    //自定义的方法，根据参数传入的匹配模式，来格式化数字
    public static void customFormat(String pattern, double value ) {
        DecimalFormat myFormatter = new DecimalFormat(pattern);
                                                        //创建模式对象
        String output = myFormatter.format(value);      //格式化数字
        System.out.println(value + "  " + pattern + "  " + output);
                                              //输出数字格式化前后的表示形式
    }
    public static void main(String[] args) {
        customFormat("###,###.###", 123456.789); //输出结果为：123,456.789
        customFormat("###.##", 123456.789);     //输出结果为：123456.79
        customFormat("000000.000", 123.78);     //输出结果为：000123.780
        customFormat("￥###,###.###", 12345.67); //输出结果为：￥12,345.67
    }
}
```

编译并运行上述程序，输出结果如下。

```
123456.789  ###,###.###  123,456.789
123456.789  ###.##  123456.79
123.78  000000.000  000123.780
12345.67  ￥###,###.###  ￥12,345.67
```

表 5.4 对输出结果进行了较为详细的解释。

表 5.4　DecimalFormatDemo.java 的输出

值	模　式	输　出	解释说明
123456.789	###,###.###	123,456.789	符号#表示一个数字，逗号是用于千分位的一个占位符，而句号是用于十进制分隔符的一个占位符
123456.789	###.##	123456.79	value 值在十进制的小数点右边有 3 位数字，但是 pattern 只有 2 位。format 方法通过四舍五入对此进行处理
123.78	000000.000	000123.780	pattern 说明数字开头和结尾的零，因为 0 字符被用于代替符号#
12345.67	$###,###.###	$12,345.67	在 pattern 中的第一个字符是美元符号（$）。注意它直接位于被格式化后的 output 的最左边

5.1.3　Math 数学运算处理类

Java 程序设计语言使用其算术运算符“+”、“–”、“*”、“/”和“%”支持基本的算术运算。在 java.lang 包中的 Math 类提供有常量，并包含有用于执行基本数学运算的方法，如初等指数、对数、平方根和三角函数等，用于进行更高级的数学计算。在 Math 类中的方法都是静态的，因此可以直接通过类来进行调用，如下所示。

```
Math.cos(angle);
```



使用静态导入（static import）的语言特性“import static java.lang.Math.*;”，就不必在每一个数学函数前都写 Math 了。这允许通过简单名称调用 Math 类的方法，例如：“cos(angle);”。

1．常量和基本方法

Math 类包含有如下两个常量。

- Math.E：代表自然对数的基数，double 类型。
- Math.PI：是圆的周长与其直径的比率（即圆周率），double 类型。

Math 类还包括超过 40 个静态方法。表 5.5 列出了其中的许多基本方法。

表 5.5　基本的 Math 方法

方　法	描　述
double abs(double d) float abs(float f) int abs(int i) long abs(long lng)	返回参数的绝对值
double ceil(double d)	返回大于等于参数的最小的整数。返回值作为 double 类型
double floor(double d)	返回小于等于参数的最大的整数。返回值作为 double 类型
double rint(double d)	返回值与参数最接近的整数。返回值作为 double 类型
long round(double d) int round(float f)	根据方法返回类型。返回与参数最接近的 long 或 int 类型值
double min(double arg1, double arg2) float min(float arg1, float arg2) int min(int arg1, int arg2) long min(long arg1, long arg2)	返回两个参数中的最小值
double max(double arg1, double arg2) float max(float arg1, float arg2) int max(int arg1, int arg2) long max(long arg1, long arg2)	返回两个参数中的最大值



【例 5.3】下面的程序 BasicMathDemo.java，详细说明了如何使用这些 Math 类及其中定义的各种静态方法。

```
public class BasicMathDemo {
    public static void main(String[] args) {
        double a = -191.635;
        double b = 41.94;
        int c = 27, d = 65;
        System.out.printf("%.3f 的绝对值是%.3f%n", a, Math.abs(a));
                                                    //运算后 a 的值为 191.635
        System.out.printf("比%.2f 大的最小整数是%.0f%n", b, Math.ceil(b));
                                                    //运算后 b 的值为 42
        System.out.printf("比%.2f 小的最大整数是%.0f%n", b, Math.floor(b));
                                                    //运算后 b 的值为 41
        System.out.printf("最接近%.2f 的整数是%.0f%n", b, Math.rint(b));
                                                    //运算后 b 的值为 42
        System.out.printf("%d 和%d 中的最大值是%d%n",c, d, Math.max(c, d));
                                                    //c 和 d 中的最大值为 65
        System.out.printf("%d 和%d 中的最小值是%d%n",c, d, Math.min(c, d));
                                                    //c 和 d 中的最小值为 27
    }
}
```

编译并运行上述程序，输出结果如下。

```
-191.635 的绝对值是 191.635
比 41.94 大的最小整数是 42
比 41.94 小的最大整数是 41
最接近 41.94 的整数是 42
27 和 65 中的最大值是 65
27 和 65 中的最小值是 27
```

2．指数和对数方法

表 5.6 列出了 Math 类的指数和对数方法。

表 5.6　指数和对数方法

方　法	描　述
double exp(double d)	返回参数的自然对数的基数 e 次幂
double log(double d)	返回参数的自然对数
double pow(double base, double exponent)	返回第一个参数的第二个参数次幂的值
double sqrt(double d)	返回参数的平方根

【例 5.4】下面的程序 ExpLogDemo.java，显示了自然对数的基数 e 的值，然后在任意选择的数字上调用在表 5.6 中列出的每一个方法。

```
public class ExpLogDemo{
    public static void main(String[] args) {
        double x = 12.715;
        double y = 3.96;
        System.out.printf("自然对数的基数 e 的值是%.4f%n", Math.E);
        System.out.printf("exp(%.3f)是%.3f%n", x, Math.exp(x));
                                                    //返回欧拉数 e 的 x 次幂的值
        System.out.printf("log(%.3f)是%.3f%n", x, Math.log(x));
```

```
                                                    //返回 x 的自然对数（底数是 e）
        System.out.printf("pow(%.3f, %.3f)是%.3f%n", x, y, Math.pow(x, y));
                                                    //返回 x 的 y 次幂的值
        System.out.printf("sqrt(%.3f)是%.3f%n", x, Math.sqrt(x));
                                                    //返回正确舍入的 x 的正平方根
    }
}
```

编译并运行上述程序，其结果如下。

```
自然对数的基数 e 的值是 2.7183
exp(12.715)是 332701.177
log(12.715)是 2.543
pow(12.715, 3.960)是 23609.827
sqrt(12.715)是 3.566
```

3．三角方法

Math 类还提供一系列的三角函数，用来进行三角运算，这些方法汇总在表 5.7 中。传递给每个方法的值都是以弧度表示的一个角度。还可以使用 toRadians()方法将角度转换为弧度。

表 5.7　三角方法

方　法	描　述
double sin(double d)	返回指定 double 值的正弦
double cos(double d)	返回指定 double 值的余弦
double tan(double d)	返回指定 double 值的正切
double asin (double d)	返回指定 double 值的反正弦
double acos(double d)	返回指定 double 值的反余弦
double atan(double d)	返回指定 double 值的反正切
double atan2(double y,double x)	将直角坐标（x,y）转换到极坐标（r,theta）并返回 theta
double toDegrees(double d) double toRadians(double d)	将参数转换为角度或弧度

【例 5.5】下面的程序 TrigonometricDemo.java 使用表 5.7 中所列的方法来计算 45 度角的各种三角函数值。其代码如下。

```
public class TrigonometricDemo {
    public static void main(String[] args) {
        double degrees = 45.0;                          //定义角度
        double radians = Math.toRadians(degrees);    //将角度转换为弧度
        System.out.format("圆周率 pi 的值是%.4f%n", Math.PI);  //输出 PI 的值
        System.out.format("%.1f 度的正弦值是%.4f%n", degrees, //输出正弦值
                                              Math.sin(radians));
        System.out.format("%.1f 度的余弦值是%.4f%n", degrees, //输出余弦值
                           Math.cos(radians));
        System.out.format("%.1f 度的正切值是%.4f%n", degrees, //输出正切值
                           Math.tan(radians));
        System.out.format("%.1f 度的双曲线余弦值是%.4f%n", degrees,
                                                    //输出双曲线余弦值
                           Math.cosh(radians));
        System.out.format("%.4f 的反正弦值是%.4f 度%n",     //输出反正弦值
                          Math.sin(radians),
                          Math.toDegrees(Math.asin(Math.sin(radians))));
```



```
        System.out.format("%.4f 的反余弦值是%.4f 度%n",      //输出反余弦值
                          Math.cos(radians),
                          Math.toDegrees(Math.acos(Math.cos(radians))));
        System.out.format("%.4f 反正切值是%.4f 度%n",        //输出反正切值
                          Math.tan(radians),
                          Math.toDegrees(Math.atan(Math.tan(radians))));
    }
}
```

编译并运行上述程序，其输出结果如下。

```
圆周率 pi 的值是 3.1416
45.0 度的正弦值是 0.7071
45.0 度的余弦值是 0.7071
45.0 度的正切值是 1.0000
45.0 度的双曲线余弦值是 1.3246
0.7071 的反正弦值是 45.0000 度
0.7071 的反余弦值是 45.0000 度
1.0000 反正切值是 45.0000 度
```

4．随机数

Math 类的 random()静态方法返回在 0.0 和 1.0 之间随机选择的一个数。区间包括 0.0 但是不包括 1.0。换句话说：0.0<=Math.random()<1.0。要在一个不同的区间获取随机值，可以在随机方法返回的值上执行算法。例如，要生成一个位于 0 和 9 之间的整数，可以像下面这样来写代码。

```
int number = (int)(Math.random()*10);
```

通过乘以 10，可能的取值范围变为 0.0<=number<10.0。

当需要生成一个单一的随机数时，使用 Math.random()就可以了。如果要生成一系列的随机数，那么应该创建一个 java.util.Random 的实例，并在此实例对象上调用方法来生成这些随机数。



5.2　字符

字符指的是用单引号括起来的单个字母。在 Java 中，表示字符的数据类型为 char。一个字符在内存中占 16 位大小的空间（2 个字节）。在编写程序的多数时候，如果想使用一个单独的字符值，通常会使用原始的 char 类型。例如：

```
char ch = 'a';
char uniChar = '\u039A';                    //大写的希腊 omega 字符的 Unicode 编码
char[] charArray = {'h','e','l','l','o'};  //一个字符的数组
char sex = '男';                             //汉字字符
```

5.2.1　Character 字符包装器类

有时候可能需要使用一个字符作为一个对象，例如，将一个字符作为一个方法的参数，而该参数应该为对象类型。同样地，Java 语言也提供了一个“包装（wrapper）器”类，用来将 char 类型的字符“包装”为一个 Character 对象。一个类型为 Character 的对象包含一个单独的字段，其类型为 char。Character 类还提供有一系列的类方法（静态方法）用于操纵字符。可以使用 Character 构造器创建一个 Character 对象，如下所示。

```
Character ch_sex = new Character('男');
```

在一定环境下，Java 编译器会根据需要自动创建一个 Character 对象。例如，如果传递一个原始的 char 类型字符到一个期望参数是对象的方法中，编译器会自动地将 char 转换为 Character。这个特性被称为“自动装箱”，或者如果转换是相反方向的，称为“拆箱”。下面是一个自动装箱的示例。

```
Character ch_sex = '男';          //原始字符'男'被装箱到 Character 对象 ch_sex 中
```

下面是一个既有装箱又有拆箱的示例。

```
Character method(Character c)  {……} //方法的参数和返回类型都是 Character 对象
char c = method ('x');               //原始的'x'被装箱用于方法 method，返回值被拆箱为字符'c'
```



Character 类是不可变的，所以一旦一个 Character 对象被创建，它就不能被改变。

表 5.8 中列出了 Character 类中最有用的一些方法（但不是全部）。Character 类的全部方法超过 50 个，请参考 API 文档中的 java.lang.Character 类。

表 5.8　Character 类中有用的方法

方　　法	描　　述
boolean isLetter(char ch) boolean isDigit(char ch)	分别用来判断指定的字符值是否是一个字母或一个阿拉伯数字
boolean isWhiteSpace(char ch)	判断指定的字符值是否是空格
boolean isUpperCase(char ch) boolean isLowerCase(char ch)	分别用来判断指定的字符值是否是大写或小写字符
char toUpperCase(char ch) char toLowerCase(char ch)	返回指定字符值的大写或小写格式
toString(char ch)	返回代表指定字符值的 String 对象，也就是只有一个字符的字符串

5.2.2　转义字符序列

在一个字符前带一个反斜线符号“\”，是一个“转义字符序列”，并且对于编译器来说，每一个转义字符序列都有一个特定的含义。在本书中的 System.out.println()语句中，已经频繁地使用到换行符“\n”这个转义字符，它的含义是在输出一个字符串后转到下一行。表 5.9 列出了 Java 中的转义序列。

表 5.9　转义字符序列

转义字符序列	描　　述
\t	在文本的当前位置处插入一个制表位
\b	在文本的当前位置处插入一个退格
\n	在文本的当前位置处插入一个新行
\r	在文本的当前位置处插入一个回车返回
\f	在文本的当前位置处插入一个走纸符
\’	在文本的当前位置处插入一个单引号字符
\"	在文本的当前位置处插入一个双引号字符
\\	在文本的当前位置处插入一个反斜杠字符



当在一个打印输出语句中遇到一个转义字符序列时，编译器会相应地对其进行解释。例如，如果想要在一个引号中带有引号时，你必须在内部的引号上使用转义字符序列 \"。例如，要输出下面的语句：

```
She said "Hello !" to me.
```

就应该像下面这样编写代码。

```
System.out.println("She said \"Hello!\" to me.");
```



5.3　使用 String 类

在 Java 语言中，把用双引号括起来零个或多个字符序列，称为字符串，如“早上好!”、“Hello World!”等。注意字符串和字符的区别，字符是用单引号括起来的一个字符，而字符串一定是用双引号括起来的零个或多个字符序列。

5.3.1　创建字符串对象

在 Java 中，把字符串作为对象来看待，不过对于初学者来说，可以把字符串看成是一种数据类型，就像其他数据类型一样，如 int 代表整数，float 代表小数，Java 用关键字 String 来代表字符串的类型。事实上，由于对于字符串的处理是如此频繁和重要，虽然在 Java 中字符串实际上是作为对象来存储的，但从使用形式上，它与一般的原始数据类型一样简单。

像其他原始数据类型一样，在使用字符串对象之前，需要先声明一个字符串变量。声明字符串变量的语法格式如下所示。

```
String 字符串变量名称;
```

例如，定义一个保存学生姓名的字符串变量，语句如下。

```
String studentName;
```

同原始数据类型一样，字符串变量必须赋初值以后才可以使用，这称为字符串对象初始化。字符串对象初始化有三种形式。

1．使用 new 运算符

前面讲过，Java 语言中，字符串实际上是以对象的形式存储的。对于以对象的形式存储的变量，需要使用 new 运算符来创建对象。例如：

```
String studentName = new String("张小明");
```

其中，studentName 为符合 Java 标识符命名规范的字符串变量名称，String 为关键字。当使用 new 运算符时，Java 自动地为字符串“张小明”分配适当大小的内存。也可以使用如下两种语句创建一个空字符串。下面这两个语句等价，都是创建一个空字符串（即不包含任何字符的字符串）。

```
String studentName1 = new String("");
String studentName2 = new String( );
```

2．直接赋初值

由于字符串处理在程序中是最经常使用的，并且非常重要，因此，Java 语言对于字符串对象的处理更为灵活，允许不使用 new 运算符就可以直接初始化。例如：

```
String studentName = "张小明";
```

在这里，等号右边的“张小明”是一个字符串字面量——在代码中用双引号括起来的一系列字符。可以直接将字符串字面量赋给 String 类型的变量。当 Java 编译器遇到字符串字面量时，它使用字符串字面量的值创建一个 String 对象，在这里是张小明。也可以创建个空字符串，然后在程序需要时再改变其值：

```
String studentName = "";
studentName = "张小明";
```

3．初始化为 null 值

对于字符串对象来说，如果一开始并无确定的初值，那么可以定义为 null，即不指向任何内容，然后在程序需要时再赋值，如下所示。

```
String studentName = null;
studentName = "张小明";
```

需要注意的是，null 值与空字符串是不同的。空字符串仅仅是不含字符，它还需要双引号括起来。而 null 值则是此变量本身就没有引用任何值。

5.3.2 获取字符串的长度

字符串对象有一个访问方法 length()，用来获取字符串的长度。调用字符串对象的 length()方法，它会返回字符串对象中所包含的字符的个数。例如下面的代码：

```
String hello = "大家好，我是张小明！";
int len = hello.length( ); //length 方法返回 hello 字符串的长度，赋给变量 len
```

调用字符串对象的方法，要使用圆点“.”运算符，如上面的代码片段中的 hello.length()，其含义代表一种从属关系，可以理解为 hello 的 length()方法。这条语句执行以后，变量 len 的值为 10。包含在字符串中的标点或空格，在计算字符串的长度时也要计算在内，如下面的代码片段：

```
String hello = "hello,my name is Jack! ";
int len = hello.length( );
```

上述代码片段执行以后，变量 len 的值为 22，包括标点符号和空格在内。

5.3.3 连接字符串

最经常对字符串进行的操作之一是将两个字符串连接起来，合并为一个长的字符串。String 类提供了一个用来连接两个字符串的方法 concat()，其使用形式如下所示。

```
string1.concat(string2);
```

concat()方法返回一个字符串，是将字符串 string2 添加到 string1 后面后形成的新字符串。如下面的代码片段：

```
String string1 = "hello, ";
String string2 = "my name is Jack! ";
String string3 = string1.concat(string2);
System.out.println(string3);
```

上面的代码片段的输出结果为：



```
"hello,my name is Jack! "
```

也可以直接使用字符串字面量来调用 concat()方法，例如下面的代码中，编译器会自动地将“hello”字符串字面量看做一个字符串对象。

```
String string3 = "hello, ".concat("my name is Jack! ");
```

连接字符串除了可以使用 concat())方法外，更经常使用的是加号“+”运算符。这是一个重载了的运算符，用来直观地连接两个字符串。例如下面的代码片段：

```
String string1 = "hello, ";
String string2 = "my name is Jack! ";
String string3 = string1 + string2;
System.out.println(string3);
```

当加号“+”用在字符串中间时，它的意思是将多个字符串合并到一起生成一个新的字符串。它使用起来比 concat()方法更加灵活。例如下面的代码，将三个字符串连接在一起。

```
String string1 = "hello, " + "my name is " + "Jack! ";
```

加号“+”运算符被广泛地用于打印输出语句 print()和 println()中，在前面的章节中，已经有大量的示例用到。例如：

```
String hello = "hello,my name is Jack! ";
int len = hello.length( );
System.out.println("字符串变量 hello 的长度为：" + len);
```

上面的代码片段的输出结果如下。

```
"字符串变量 hello 的长度为：22
```

在输出语句中，加号“+”运算符同样可以用来连接多个字符串，例如下面的代码中，使用“+”运算符将字符串和整数类型变量混合连接起来输出。

```
String name = "张小明";
int age = 18;
System.out.println(name + "的年龄是" + age);
```

在输出语句中，将使用变量 name 和 age 的值参与字符串连接。当其他原始数据类型与字符串对象进行加法运算时，Java 会自动地将它们转换为字符串对象，然后进行连接。

需要注意的是，如果表达式中包含了多个加号“+”运算符，并且存在各种数据类型参与运算，则按照加号“+”运算符从左到右地进行运算，Java 会根据加号“+”运算符两边的操作数类型来决定是进行算术运算还是字符串连接的运算。例如：

```
System.out.println(100 + 32.5 + "RMB");     //先进行算术运算，再进行字符串连接
System.out.println("RMB" +100 +32.5);       //进行字符串连接
```

对于上述代码片段的第一行代码，按照加号“+”从左至右的结合性，先计算“100+32.5”，结果为 132.5。然后计算“132.5+"RMB"”，结果为“132.5RMB”；对于第二行代码，先计算“"RMB"+100”，结果为字符串“RMB100”，然后再计算“RMB100+32.5”，运算结果为“RMB10032.5”。

也可以使用加号“+”运算符来将字符串与任何对象进行混合运算。对于任何一个非 String 的对象，会调用它们的 toString()方法来将其转换为字符串。

在 Java 语言中，不允许在源文件中将一个字符串字面量分为多行，所以程序员必须在一个多行显示的字符串字面量的每一行结尾使用加号“+”连接运算符。例如下面的代码片段：

```
String address = "张小明的家乡是中华人民共和国广西壮族自治区" +
                 "宜州市龙头乡老龙头村三组";
```

在多行之间使用加号“+”运算符进行连接，以使代码更加清晰并易于阅读，这在打印输出语句中使用很普遍。

```
System.out.println("姓名：" + name +
                   "年龄：" + age +
                   "家庭住址：" + address);
```

5.3.4 字符串与字符数组

可以方便地将字符数组转换为字符串，然后利用字符串对象的属性和方法，对其进行进一步处理。将字符数组转换为字符串的代码形式如下。

```
char[ ] helloArray = {'h','e','l','l','o','.'};
String helloString = new String(helloArray);    //将字符数组作为构造函数的参数
System.out.println(helloString);
```

上面的代码片段输出“hello.”。在使用 new 运算符创建字符串对象时，将字符数组作为构造函数（将在“类和对象”一章讲解）的参数，可以将字符数组转换为字符相应的字符串。

相反，也可以将字符串转换为字符数组，这需要使用字符串对象的一个方法 toCharArray()。它返回一个字符串对象转换过来的字符数组。

【例 5.6】字符串与字符数组转换示例程序。在程序中定义一个字符串变量，通过调用字符串对象的 toCharArray()方法，可以将字符串对象转换为字符数组。

```
public class StringToChar{
    public static void main(String[ ] args){
        String helloString = "hello! ";          //声明一个字符串变量并赋初值
        char[ ] helloArray = helloString.toCharArray( ); //将字符串转换为字符数组
        for(int i=0;i<helloArray.length;i++){    //循环遍历字符数组，并输出数组当中每一个字符元素
            System.out.println(helloArray[i]);
                                                 }
                                             }
                                         }
```

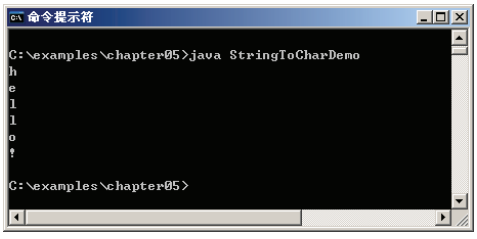


图 5.2 字符串转换为字符数组

上述代码将依次输出字符串“hello!”当中的每一个字符。编译并运行此程序，其输出结果如图 5.2 所示。

【例 5.7】下面的示例用来统计一行字符中字母、数字及其他字符的个数。程序代码如下所示。

```
// 统计一行字符中字母、数字以及其他字符的个数
public class  CountCharDemo{
    public static void main(String[ ] args){
        String string1 = "hello,my name is Jack,I am 18 years! ";
        int count1 = 0,count2 = 0,count3 = 0;           //分别用来存放字母、数字和其他字符的个数
        char[ ] charArray = string1.toCharArray( );     //将字符串变量转换为
```



```
字符数组
            for(int i=0;i< charArray.length;i++){              //遍历字符数组
                //判断字符数组中每一个字符元素的类型
                if(charArray[i]<='z' && charArray[i]>= 'a' || charArray[i]<= Z'
&& charArray[i]>='A')
                    count1++;                                  //如果是字母字符，count1 增 1
                else if(charArray[i]<='9' && charArray[i]>='0')   //如果是数字字符
                    count2++;                                  //计数变量 count2 增 1
                else                                           //如果既不是字母又不是数字
                    count3++;                                  //计数变量 count3 增 1
            }
            System.out.println("含有字母数量：" + count1);      //输出字母字符的个数
            System.out.println("含有数字数量：" + count2);      //输出数字字符的个数
            System.out.println("含有其他字符数量：" + count3); /输出其他字符的个数
        }
    }
```

编译并运行例 5.7，输出结果如图 5.3 所示。

设计思路为：将字符串转换为字符数组，然后遍历字符数组。如果字符的 ASCII 码值在小写字母“z”和小写字母“a”之间，或者在大写字母“Z”和大写字母“A”之间，则字母计数变量增 1；如果字符的 ASCII 码值在数字字符 0～9 之间，则数字计数变量增 1；否则，其他计数变量增 1。

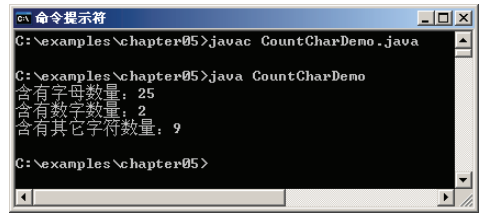


图 5.3　字母、数字与字符统计

5.3.5　在一个字符串中操纵字符

String 类提供了大量的方法来检查字符串的内容，例如，在一个字符串中查找特定的字符或子字符串，以及改变字符串中的内容等。下面就介绍这些在字符串中操纵字符的方法。

1．通过索引获得字符和子字符串

通过调用 charAt()方法，可以在一个字符串中获得特定索引处的字符。字符串中第一个字符的索引是 0，同时最后一个字符的索引是 length()–1。例如，下面的代码获得一个字符串的索引为 7 的字符。

```
String string1 = "Java programming! ";
char aChar = string1.charAt(7);
```

索引是从 0 开始的，因此索引为 7 处的字符是“O”，如图 5.4 所示。

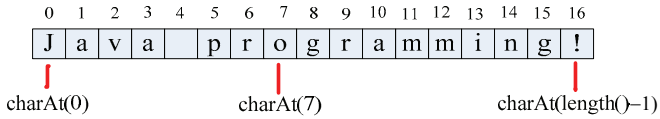


图 5.4　charAt()方法

如果想从一个字符串中获得超过一个字符的连续字符序列（即获得字符串的一个子字符串），可以使用 substring()方法。substring()方法有两个版本，如表 5.10 所示。

表 5.10　String 类的 substring()方法

方　法	描　述
String substring(int beginIndex, int endIndex)	返回字符串中从 beginIndex 到 endIndex（不含 endIndex）之间的字符序列
String substring(int beginIndex)	返回字符串中从 beginIndex 到字符串结尾的字符序列

请看下面的示例，从字符串“Java programming!”中获取从索引为 8 到索引为 12（但不包含索引 12）的子字符串“gram”，并赋给字符串变量 string1。字符串对象 string1 在内存中的状态如图 5.5 所示。

```
String string1 = "Java programming! ";
char aChar = string1.substring(8,12);
```

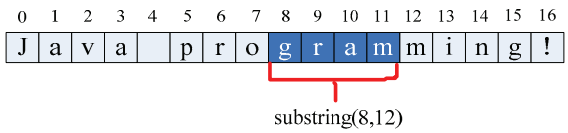


图 5.5　String 类的 substring()方法

如果改用第二种方法形式 substring(int geginIndex)，那么获取的子字符串为“gramming!”，如以下代码所示。

```
String string1 = "Java programming! ";
char aChar = string1.substring(8);      //返回从索引 8 处到字符串结尾的字符序列
```

2．操纵字符串的方法

String 类还有许多其他的操纵字符串的方法，如分隔字符串、清除字符串两端的空格、大写字符串转换为小写字符串，或小写字符串转换为大写字符串等，如表 5.11 所示。

表 5.11　用于操纵字符串的 String 类的其他方法

方　法	描　述
String[] split(String regex) String[] split(String regex, int limit)	按照参数（包含有正则表达式）所指定的规则，将字符串分割为一个相应的字符串数组。可选的第二个整型参数指定了返回数组的最大长度
CharSequence subSequence(int beginIndex, int endIndex)	返回字符串中从 beginIndex 直到 endIndex−1 的新的字符序列
String trim()	返回移除了开头和结尾处空格的字符串的拷贝
String toLowerCase() String toUpperCase()	返回转换为小写或大写以后的字符串的拷贝。如果没有必要的转换，这些方法返回原始的字符串

【例 5.8】操纵字符串方法示例。

```
public class StringMethods{
    public static void main(String[ ] args){
        String originalString = "hello,my name is Jack,I am 18 years!";
        System.out.println("原始字符串为: \"" + originalString + "\"");
        String[ ] splitedString = originalString.split(",");
                                                    //使用逗号分隔符拆分字符串
        System.out.print("按逗号(,)分隔以后的字符串数组元素为: ");
        for(int i=0;i<splitedString.length;i++){
                                                    //循环输出拆分后的字符串中的元素
            System.out.print("[" + splitedString[i] + "]");
        }
```



```
        System.out.println();
        splitedString = originalString.split("m");  //使用字符'm'拆分字符串
        System.out.print("按字母'm'分隔以后的字符串数组元素为：");
        for(int i=0;i<splitedString.length;i++){     //循环输出拆分后的字符串
中的元素
            System.out.print("[" + splitedString[i] + "]");
        }
        System.out.println();
        System.out.print("原始字符串从第 2 个到第 5 个的字符序列为：");
        System.out.println("\"" + originalString.subSequence(2,6) + "\"");
                                                //输出子字符串
        System.out.print("原始字符串转换为大写字母以后：");
        System.out.println("\"" + originalString.toUpperCase() + "\"");
                                                //将原字符串全部转换为大写
        System.out.print("原始字符串转换为小写字母以后：");
        System.out.println("\"" + originalString.toLowerCase() + "\"");
                                                //将原字符串全部转换为小写
    }
}
```

编译并运行上述程序，输出结果为如图 5.6 所示。

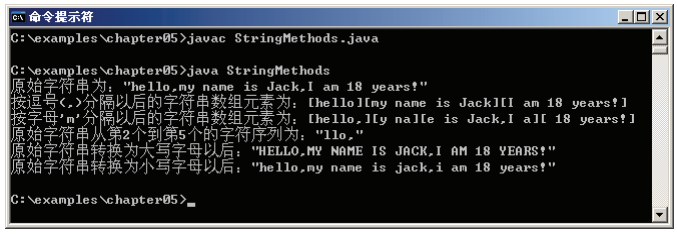


图 5.6　操纵字符串的各种方法

需要注意的是，大部分操纵字符串的方法，仅仅是获得原始字符串的一个拷贝，即改变并不影响原始字符串的内容。

3．在一个字符串中查找特定字符或子字符串

String 类还提供了一些方法，用来在一个字符串中查找指定的字符或子字符串，或指定的字符或子字符串出现的位置，如表 5.12 所示。

表 5.12　各种字符串的查找方法

方　法	描　述
int indexOf(int ch) int lastIndexOf(int ch)	返回指定字符第一次（或最后一次）出现的索引值
int indexOf(int ch, int fromIndex) int lastIndexOf(int ch, int fromIndex)	从指定的索引位置从前往后（或从后往前）开始查找，返回指定的字符第一次（或最后一次）出现的索引值
int indexOf(String str) int lastIndexOf(String str)	返回指定子字符串第一次（或最后一次）出现的索引值
int indexOf(String str, int fromIndex) int lastIndexOf(String str, int fromIndex)	从指定的索引位置从前往后（或从后往前）开始查找，返回指定的子字符串第一次（或最后一次）出现的索引值
boolean contains(CharSequence s)	如果字符串包含指定的子字符串（字符序列），则返回 true；否则，返回 false

表 5.12 中，indexOf()和 lastIndexOf()方法返回指定字符或子字符串在原字符串中的位

置。其中 indexOf()方法从字符串的起始索引位置开始，从前往后查找，lastIndexOf()方法从字符串的最后索引位置开始，从后往前查找。如果没有找到指定的字符或子字符串，indexOf()和 lastIndexOf()方法都返回–1。

当需要判断字符串是否包含一个字符序列，但是不需要知道详细位置时，使用 contains()方法。如果字符串包含一个特定的字符序列，这个方法返回 true。需要注意的是，contains()方法的参数是一个 CharSequence 类型的对象，而 CharSequence 是 String 类实现的一个接口，因此 contains()方法的参数也可以是一个 String 字符串对象。

【例 5.9】查找字符串的方法示例。

```
public class FindCharDemo{
    public static void main(String[ ] args){
        //判断电子邮箱地址是否正确
        String email = "zhangxiaoming@163.com"; //合法的邮箱地址
        int index = email.indexOf('@');          //判断邮箱地址中是否包含字符@
        if(index != -1)                          //说明邮箱地址中有@符号
            System.out.println("您的邮箱正确！");
        else                                     //如果没有找到@符号
            System.out.println("您的邮箱不正确！");
        //分析字符串，获得文件名
        String url = "C:\\examples\\chapter05\\Test.java";
                                                 //用于测试的文件路径名
        int lastIndex = url.lastIndexOf('\\');   //查找最后一个文件分隔符的位置
        if(lastIndex != -1){
            String filename = url.substring(lastIndex+1);
                                                 //截取路径中的文件名
            System.out.println("您要查找的文件为：" + filename);
        }
    }
}
```

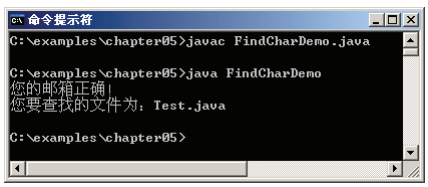


图 5.7　查找字符和子字符串

编译并运行上述程序，运行结果如图 5.7 所示。

在程序 FindCharDemo 中，判断电子邮箱地址是否正确。正确的电子邮箱地址必须包含一个'@'符号，因此使用 indexOf()方法查找电子邮箱中是否包含'@'符号。如果返回–1，说明没有找到'@'符号，此电子邮箱为不合法的邮箱。在实际应用中，可能还包含更多的判断，如是否包含小圆点，以'@'符号不能为字符串的第一个字符等，这都可以使用 String 类所提供的方法来判断。

4．替换一个字符串中的字符和子字符串

有时候，需要用指定的字符或字符串替换字符串中特定的字符或字符串，如 Word 中的查找和替换的功能。String 类有 4 个方法用来替换指定的字符和子字符串，如表 5.13 所示。

表 5.13　替换字符串中的字符和子字符串

方　法	描　述
String replace(char oldChar, char newChar)	使用 newChar 字符替换字符串中当前所有的 oldChar 字符，并返回替换后的一个新字符串（原字符串不受影响）

续表

方　法	描　述
String replace(charSequence target, charSequence replacement)	使用指定的 replacement 字符序列替换字符串中的与 target 字符序列相匹配的子字符串（原字符串不受影响）
String replaceAll(String regex, String replacement)	使用指定的 replacement 字符串替换原字符串中每一个与 regex 字符串相匹配的子字符串（原字符串不受影响）
String replaceFirst(String regex, String replacement)	使用指定的 replacement 字符串替换原字符串中第一个与 regex 字符串相匹配的子字符串（原字符串不受影响）

【例 5.10】替换字符串中字符和子字符串的方法示例。

```
public class ReplaceDemo{
    public static void main(String[ ] args){
        //判断电子邮箱地址是否正确
        String email = "zhangxiaoming@163.com";
        //String newEmail = email.replace("163","126");//替换相匹配的子字符串
        String newEmail = email.replace('z','c');    //使用字符'c'替换指定的
字符'z'
        System.out.println("email:"+email);          //原字符串不受影响
        System.out.println("newEmail:"+newEmail);
    }
}
```

编译并运行上述程序，输出结果如图 5.8 所示。

在程序 ReplaceDemo 中，使用字符串"126"替换原电子邮箱中的"163"，使用字符'c'替换指定的字符'z'。

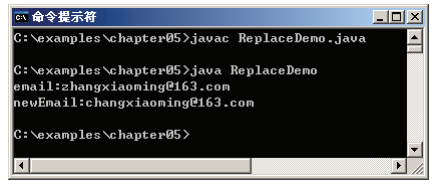


图 5.8　字符和字符串替换

5.3.6　字符串比较和字符串部分内容比较

String 类还提供了一系列的方法，用来对字符串进行比较或对字符串的部分内容进行比较，例如比较两个字符串是否相等、字符串是否以指定的前缀开始或以指定的后缀结尾等。这些方法详见表 5.14。

表 5.14　用于字符串比较的方法

方　法	描　述
boolean endsWith(String suffix) boolean startsWith(String prefix)	如果字符串以指定的子字符串（作为方法的参数）结束或开始，返回 true，否则返回 false
boolean startsWith(String prefix, int offset)	从索引 offset 开始，如果字符串以指定的子字符串（作为方法的参数）结束或开始，返回 true，否则返回 false
int compareTo(String anotherString)	按字典顺序比较两个字符串，返回一个整数。如果大于、等于、小于参数字符串，则分别返回大于 0 的整数、0、小于 0 的整数
int compareToIgnoreCase(String str)	按字典顺序比较两个字符串，忽略大小写，返回一个整数。如果大于、等于、小于参数字符串，则分别返回大于 0 的整数、0、小于 0 的整数
boolean equals(Object anObject)	如果参数字符串对象的字符序列与当前字符串的对象序列完全相同（即内容相同），则返回 true；否则返回 false
boolean equalsIgnorecase(String anotherString)	如果参数字符串对象的字符序列与当前字符串的对象序列忽略大小写后完全相同（即内容相同），则返回 true；否则返回 false

续表

方　法	描　述
boolean regionMatches(int toffset,String other,int ooffset,int len)	测试当前字符串指定范围与参数字符串指定范围是否相匹配。指定范围为当前字符串从索引 toffset 开始长度为 len 的字符序列和参数字符串从索引 ooffset 开始长度为 len 的字符序列
boolean regionMatches(boolean ignoreCase,int toffset,String other,int ooffset,int len)	同上一方法。其中第一个参数为 true 时，比较字符串时忽略大小写
boolean matches(String regex)	测试当前字符串是否与参数正则表达式相匹配

【例 5.11】使用 equals()方法对两个字符串进行比较。

```
public class StringEqualsDemo{
    public static void main(String[ ] args){
        String email1 = "zhangxiaoming@163.com";
        String email2 = "zhangxiaoming@163.com";
        String email3 = new String("zhangxiaoming@163.com");
        boolean isEquals1 = email1.equals(email2);      //将字符串 email1 与
email2 进行内容比较
        boolean isEquals2 = (email1==email2);           //将字符串 email1 与
email2 进行地址比较
        boolean isEquals3 = email2.equals(email3);      //用来比较字符串的内
容，所以返回 true
        boolean isEquals4 = (email2==email3);           //用来比较字符串的地
址，所以返回 false
        System.out.println("isEquals1: " + isEquals1);
        System.out.println("isEquals2: " + isEquals2);
        System.out.println("isEquals3: " + isEquals3);
        System.out.println("isEquals4: " + isEquals4);
    }
}
```

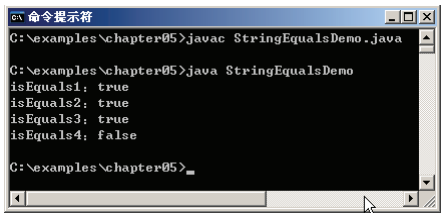


图 5.9　字符串比较

编译并运行上述程序，输出结果如图 5.9 所示。

注意上例中，字符串 email1 和 email2 都没有使用 new 运算符，Java 会自动地将同一个字符串对象赋给这两个变量，所以 email1 和 email2 在内存中的地址和内容都是相同的。而 email3 使用了 new 运算符来创建，所以 Java 为 email3 重新分配内存，即使它包含的字符序列与 email1 完全相同，它与 email1 也是两个独立的字符串对象。

所以在用 equals()方法进行比较时，仅比较两个字符串的内容，因此返回值都为 true；而使用“==”运算符进行比较时，实际上是对变量内存地址的比较，所以返回 false。在比较两个字符串时，通常都是对内容进行比较，不能使用操作符“==”。

【例 5.12】使用字符串的 regionMatches()方法对字符串部分内容进行比较。

```
public class RegionMatchesDemo {
    public static void main(String[] args) {
        String searchMe = "Green Eggs and Ham"; //声明字符串变量 searchMe
        String findMe = "Eggs";                 //声明字符串变量 findMe
        int searchMeLength = searchMe.length(); //获得 searchMe 字符串的长度
```

```
        int findMeLength = findMe.length();      //获得 findMe 字符串的长度
        boolean foundIt = false;
        for (int i = 0; i <= (searchMeLength - findMeLength); i++) {
//遍历字符串进行比较
            if (searchMe.regionMatches(i, findMe, 0, findMeLength)) {
                foundIt = true;
                System.out.println(searchMe.substring(i, i + findMeLength));
                break;
            }
        }
        if (!foundIt) System.out.println("没有发现相匹配的内容.");
    }
}
```

在上述程序中，使用了字符串的一个方法 regionMatches()，该方法有 4 个参数，第 1 个参数代表从源字符串开始进行比较的位置，第 2 个参数代表要进行比较的子字符串，第 3 个参数代表从子字符串开始进行比较的索引位置，第 4 个参数代表子字符串的长度。编译并运行程序，输出结果如图 5.10 所示。

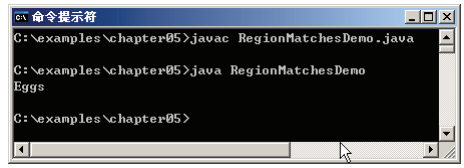


图 5.10　匹配部分字符串内容

【例 5.13】使用 startsWith()和 endWith()方法进行简单的身份验证的示例程序。

```
public class IdentifyCheck {
    public static void main(String[ ] args) {
        String ID = new String("410212198502130628");   //定义变量 ID，保存身份证号码
        String name = "先生";
        String address = "未知";
        if(ID.startsWith(    "410")){                   //如果身份证以“410”开头
            address = "郑州市";
    }
        //判断性别，单数为男性，偶数为女性
        for(int i=0;i<10;i++,i++){                      //循环所有的偶数（个位）
            String endChar = ""+i;                      //将一位偶数变为字符串
            if(ID.endsWith(endChar)){                   //如果 ID 以偶数结尾
                name = "女士";                          //改变称呼
                break;
            }
        }
        System.out.println(name + ",您的地址：  " +address);  //输出信息
    }
}
```

编译并运行程序，输出结果如下。

“女士，您的地址：郑州市”

在程序 IdentifyCheck 中，使用 String 类的 startsWith()方法，判断代表身份证号的 ID 是否以指定的参数开头。身份证的头三位代表了地址。使用 String 类的 endsWith()方法，判断代表身份证号的 ID 是否以指定的参数结尾，并进而根据最后一位数的奇偶性判断性别。如果以奇数结尾，则为男士，如果以偶数结尾，则为女士。

5.4 使用 StringBuilder 类

一个 String 对象的长度是固定的，不能改变它的内容，也不能附加新的字符至 String 对象中。可以使用加号“+”运算符来连接字符串以达到附加新字符或字符串的目的，但使用加号“+”运算符会产生一个新的 String 实例，即需要另外分配空间。如果既想节省开销，又想改变字符串的内容，则可以使用 StringBuilder 类。

5.4.1 认识 StringBuilder 类

J2SE 5.0 提供 java.lang.StringBuilder 类，使用这个类所产生的对象默认会有 16 个字符的长度，程序员也可以自行指定初始长度。StringBuilder 类就像一个可变长度的字符数组一样，内容和长度都可以被改变。如果附加的字符超出可容纳的长度，则 StringBuilder 对象会自动增加长度以容纳被附加的字符。

【例 5.14】String 与 StringBuilder 类的比较。

```
public class StringAndStringBuilder {
    public static void main(String[] args) {
        String textString1 = new String("java"); //在内存中为变量 textString1
分配内存空间
        String textString2 = textString1+ 'c';  //在内存中为变量 textString2
重新分配空间
        StringBuilder builderString1 = new StringBuilder("java");   //在内
存中为变量 builderString1 分配空间
        StringBuilder builderString2 = builderString1.append("c"); //在原
字符串后面追加字符'c'
        System.out.println(textString1 + "  " + textString2);
        System.out.println(builderString1 + "  " + builderString2);
    }
}
```

程序执行以后的输出结果如图 5.11 所示。

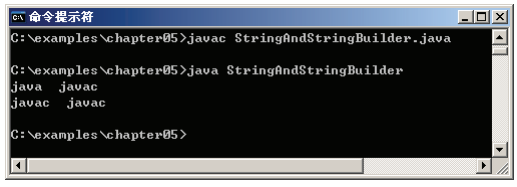


图 5.11　String 对象和 StringBuilder 对象的比较

上例中的字符串对象 textString1、textString2 在内存中的变化如图 5.12 所示。

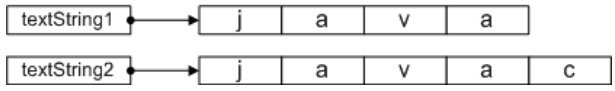


图 5.12　String 对象在内存中的变化

builderString1、builderString2 在内存中的变化如图 5.13 所示。

从上面的内存变化可以看出，String 类型和 StringBuilder 类型的主要性能区别其实在于



String 是不可变的对象，因此在每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象。所以经常改变内容的字符串最好不要用 String，因为每次生成对象都会对系统性能产生影响，特别是当内存中无引用的对象多了以后。

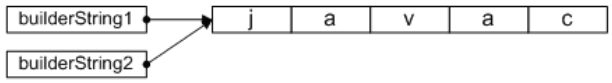


图 5.13　StringBuilder 对象在内存中的变化

而如果是使用 StringBuilder 类则结果就不一样了，每次结果都会对 StringBuilder 对象本身进行操作，而不是生成新的对象后再改变对象引用。所以在一般情况下推荐使用 StringBuilder 类型，特别是字符串对象经常改变的情况下，例如有频繁做字符串附加的需求。使用 StringBuilder 类型会让程序的效率大大提高。

5.4.2　StringBuilder 对象的长度和容量

StringBuilder 类有一个 length()方法，调用它会返回在 StringBuilder 字符串对象中的字符序列的长度，就像 String 类一样。但是 StringBuilder 类还有一个 String 类所没有的方法 capacity()。capacity()方法可返回该对象目前已经被分配的、可容纳的字符容量，它总是大于或等于字符串对象的长度，并且根据情况可自行进行扩展。请看下面的代码片段：

```
StringBuilder sb = new StringBuilder();    //创建一个空的 StringBuilder 对象，初始容量为 16
sb.append("Greetings");                    //在内存空间的开始添加 9 个字符
```

上述两行代码创建了个容量为 16 而长度为 9 的一个 StringBuilder 对象 sb，如图 5.14 所示。

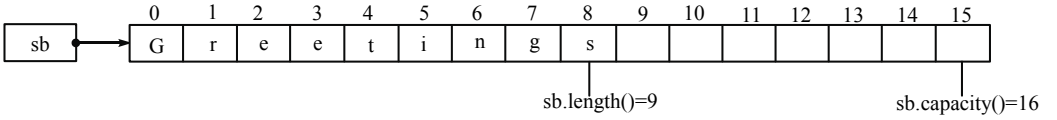


图 5.14　容量为 16 长度为 9 的 StringBuilder 对象

StringBuilder 类有一些 String 类所不具备的与长度和容量有关的方法，如表 5.15 所示。

表 5.15　与长度和容量有关的方法

方　法	描　述
void setLength(int newLength)	设置字符序列的长度。如果 newLength 比 length()方法得到的长度小，那么超过部分的字符将会被截去。如果 newLength 比 length()方法得到的长度大，那么多出的内存空间将添加上 null 字符
void ensureCapacity(int minCapacity)	确保容量至少等于指定的最小容量值

5.4.3　StringBuilder 类提供的操作方法

在 StringBuilder 对象上所具有而在 String 对象上所没有的主要操作，是 append()和 insert()方法，它们有多种重载的形式，可以把不同类型的数据转化为字符序列，然后添加或插入到 StringBuilder 对象中。append()方法总是添加这些字符串到已经存在的字符序列的最后，而 insert()方法则将字符或字符串添加到指定的位置。关于 StringBuilder 类的各种

方法，如表 5.16 所示。

表 5.16　各种 StringBuilder 方法

方　法	描　述
StringBuilder append(boolean b) StringBuilder append(char　c) StringBuilder append(char[] str) StringBuilder append(char[] str,int offset,int len) StringBuilder append(double d) StringBuilder append(float f) StringBuilder append(int i) StringBuilder append(long lng) StringBuilder append(Object obj) StringBuilder append(String s)	添加参数到 StringBuilder 对象中。在追加操作发生前，数据首先被转换为一个字符串
StringBuilder delete(int start, int end) StringBuilder deleteCharAt(int index)	删除 StringBuilder 对象中指定的字符或字符序列
StringBuilder insert(int offset, boolean b) StringBuilder insert(int offset, char c) StringBuilder insert(int offset, char[] str) StringBuilder insert(int index, char[] str, int offset, int len) StringBuilder insert(int offset, double d) StringBuilder insert(int offset, float f) StringBuilder insert(int offset, int i) StringBuilder insert(int offset, long lng) StringBuilder insert(int offset, Object obj) StringBuilder insert(int offset, String s)	将第二个参数指定的内容插入到 StringBuilder 对象中。第一个参数指明在这个索引前面插入数据。在执行插入操作之前，这些被插入的数据首先被转换成字符串
StringBuilder replace(int start, int end, String s) void setCharAt(int index, char c)	在 StringBuilder 对象中替换指定的字符或字符序列
StringBuilder reverse()	翻转 StringBuilder 对象中的字符序列
String toString()	返回一个包含在 StringBuilder 中的字符序列的字符串

如果想使用 String 类的方法来处理 StringBuilder 对象，可以先使用方法 toString()方法将一个 StringBuilde 对象转换为一个 String 对象，然后就可以使用 String 类的方法了；也可以使用 StringBuilder(String str)构造函数将 String 对象转换为 StringBuilder 对象。

【例 5.15】使用 String 对象实现输出一首回文诗。

```
public class StringDemo {
    public static void main(String[] args) {
        String palindrome = "香莲碧水动风凉 水动风凉夏日长";
        int len = palindrome.length();                  //获得字符串对象的长度
        char[] tempCharArray = new char[len];           //创建字符数组
        char[] charArray = new char[len];               //创建字符数组
        //将原始字符串中的字符放到字符数组中
        for (int i = 0; i < len; i++) {
            tempCharArray[i] = palindrome.charAt(i);
        }
        //反转字符数组中的字符
        for (int j = 0; j < len; j++) {
            charArray[j] = tempCharArray[len - 1 - j];
        }
```



```
        String reversePalindrome = new String(charArray);
                                          //使用反转后的字符数组创建新的字符串
        System.out.println(palindrome);
        System.out.println(reversePalindrome);
    }
}
```

编译并运行上述程序，输出结果如图 5.15 所示，这是明末浙江才女吴绛雪《四时山水诗》中的夏景诗。



图 5.15　使用 String 类的方法实现字符串的翻转

在上例中，要完成字符串的翻转，程序将字符串转换到一个字符数组（第一个 for 循环），反转次序将字符复制到另一个数组中（第二个 for 循环），然后再重新生成新的字符串，并输出。

上例中使用的是 String 对象，如果在程序中改用 StringBuilder 对象，那么就可以使用其 reverse()方法，这将使代码更加简洁并且易读。

【例 5.16】使用 StringBuilder 对象实现输出一首回文诗。

```
public class StringBuilderDemo {
    public static void main(String[] args) {
        String palindrome = "香莲碧水动风凉 水动风凉夏日长";
        StringBuilder sb = new StringBuilder(palindrome); //创建StringBuilder对象
        sb.reverse();                                     //翻转字符串
        System.out.println(palindrome);                   //输出原来的字符串内容
        System.out.println(sb);                           //输出翻转以后的字符串内容
    }
}
```



图 5.16　使用 StringBuilder 类的方法实现字符串的翻转

输出结果如图 5.16 所示（与图 5.15 的输出结果相同）：

在程序 StringBuilderDemo 中，将文字顺序的颠倒，改为使用 StringBuilder 对象的 reverse()方法来实现。可以看出，这样处理的方式很简单，使程序更加简洁。

5.5　实例

关于字符串的处理，在实际开发应用程序时会经常用到。本节就通过两个应用实例“用户登录验证程序”和“敏感词过滤程序”，帮助读者了解和掌握字符串处理在实际应用程序开发中的应用。这两个实例都用到了本节所讲述的字符串处理方法。

5.5.1　实例 1：用户登录验证程序

在很多程序中，为了保护数据信息的安全，需要对登录应用程序的用户进行验证，只有通过验证的用户，如已经注册的用户或指定权限的用户，才能进入系统并拥有相应的操作权限。

最经常采用的用户登录验证方式是对用户所输入的用户名和密码进行验证。如果用户

输入的用户名和密码与保存在数据库中或文件中的原始注册信息相吻合，就认为是合法用户，从而允许其进入系统，用户就具有相应的对系统的操作权限；否则，就认为其是非法用户，拒绝其登录。请看下面这个进行用户登录验证的程序，在这里只对用户输入的用户名和密码进行验证。

```
//用户登录验证
public class LoginCheck{
    public static void main(String[] args){
        String originalUserName = "zhangxiaoming@163.com";   //声明固定用户名
        String originalPassword = "123456";                  //声明原始密码
        String userName,userPwd;
        //判断是否输入了用户名和密码
        if(args.length<2){                              //如果命令行参数数组长度小于 2
            System.out.println("请输入用户名和密码！");
            return;                          //立即结束程序并返回，后面的代码不再执行
        }
        //获得用户输入的用户名和密码
        userName = args[0].trim();       //trim()方法消除字符串首尾的空格
        userPwd = args[1].trim();
        //将用户输入了用户名和密码与原始的用户名和密码进行比较
        if(!userName.equals(originalUserName)){     //如果用户输入用户名与程
序中固定用户名不同
            System.out.println("抱歉，您的用户名不正确！请重新输入");
        }else if(!userPwd.equals(originalPassword)){ //如果用户输入的密码与程
序中原始密码不同
            System.out.println("抱歉，您的密码不正确！请重新输入");
        }else{                                          //如果用户名和密码都正确
            int index = originalUserName.indexOf('@');   //查找用户名中有没有
符号'@'
            String name = originalUserName.substring(0,index);
                                                        //截取'@'前的用户名
            System.out.println(name + ",欢迎您！您已经通过验证，可以进行操作！");
        }
    }
}
```

程序中，设定了原始的用户名“zhangxiaoming@163.com”和密码“123456”，在实际开发过程中，原始的用户名和密码应该来源于数据库或者文件或者其他保存数据的地方。在本例中进行了简化，将用户名和密码设置为固定的值，编码在程序中了。程序中，用来判断用户登录时是否输入了用户名和密码的代码如下。

```
if(args.length<2){
    System.out.println("请输入用户名和密码！");
    return;                                  //立即结束程序返回，后面的代码不再执行
}
```

这里的 args 是 main()方法的字符串数组参数。它自动保存用户在执行 Java 程序时所输入的参数字符串序列。例如下面的代码为执行此程序的命令及参数。

```
java LoginCheck  "张小明" "hello world! " "are you ready? "
```

在执行程序的命令 Java 后面所跟的第一个字符串为程序的名字，第二个字符串以后都



是参数字符串，这里有三个，分别是“张小明”、“hello world!”、“are you ready”。这三个参数字符串都会自动被保存到 main()方法的字符串参数 args 中。然后在程序中，可以通过数组 args 的下标索引来引用这些字符串。在本例中，用户需要在命令行中输出如下命令：

```
java LoginCheck  "zhangxiaoming@163.com" "123456"
```

这样代表用户名和密码的字符串“zhangxiaoming@163.com”、“123456”会自动地保存到参数数组 args 中，在程序中就可以通过下标索引引用它们。注意这里的“return;”语句。当在程序当中执行“return;”语句时，程序会立即终止并返回，后面的代码不再执行。在程序中，使用下面的代码来获取用户名和密码。

```
userName = args[0].trim();                    //trim()方法消除字符串首尾的空格
userPwd = args[1].trim();
```

其中 args[0]是参数字符串数组的第一个元素，也就是用户输入的用户名，而 args[1]就是参数字符串数组的第二个元素，即用户输入的密码。trim()为 String 类的方法，用来消除字符串首尾的空格。在进行用户名和密码验证时，为了防止用户无意中输入多余的空格而无察觉，需要消除这种误操作而产生的空格。

最后，调用 String 类的 equals()方法，将用户登录时输入的用户名和密码与程序中保存的原始用户名和密码进行比较，如果相符，用户登录成功；否则，给出相应的提示信息。输出结果如图 5.17 所示。

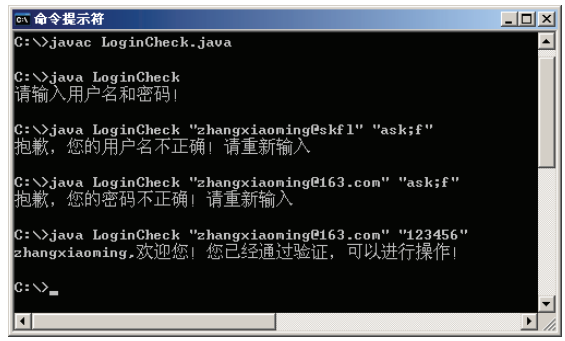


图 5.17　用户登录验证

5.5.2　实例 2：敏感词过滤程序

在网络程序中，如聊天室、聊天软件等，经常需要对一些用户所提交的聊天内容中的敏感性词语进行过滤。如“性”、“色情”、“爆炸”、“恐怖”、“枪”等，这些都不可以在网上进行传播，需要过滤掉或者用其他词语替换掉。

一种简单的敏感词过滤的方法，是将用户的聊天内容保存到一个字符串对象或一个 StringBuilder 对象中，然后与敏感词语进行比对。如果属于敏感词语，就过滤掉或替换掉。请看下面的这个程序，实现了简单的敏感词过滤功能。

```
//敏感词过滤
public class FiltWords{
    public static void main(String[] args){
        //设置敏感词
        String[ ] sensitiveWords1 = {"性","色情","爆炸","恐怖","枪"};
//需要屏蔽的敏感词
        String[ ] sensitiveWords2 = {"暴力","拉登","导弹"};
//需要被替换的敏感词
        //接收用户通过控制台传进来的参数字符串，模拟用户的聊天内容
        if(args.length<1){
            System.out.println("没有输入聊天内容，请重新输入！");
```

```
            return;                    //立即结束程序并返回，后面的代码不再执行
        }
        String chatContent = args[0];
        //将用户聊天内容与需要屏蔽的敏感词进行匹配
        for(int i=0;i<sensitiveWords1.length;i++){
            int index = chatContent.indexOf(sensitiveWords1[i]);
//查找要屏蔽的敏感词
            if(index != -1)          //如果查找到了敏感词
               chatContent = chatContent.replaceAll(sensitiveWords1[i],"");
                                                    //以空字符串屏蔽掉敏感词
        }
        //将用户聊天内容中需要替换的敏感词替换掉
        for(int i=0;i<sensitiveWords2.length;i++){
            int index = chatContent.indexOf(sensitiveWords2[i]);
                                                    //查找要替换的敏感词
            if(index != -1)                         //以指定字符串替换掉敏感词
                chatContent = chatContent.replaceAll(sensitiveWords2[i],"和平");
        }
        //输出过滤以后的聊天内容
        System.out.println("您的聊天内容为:" + chatContent);
    }
}
```

在程序中，定义了两个敏感词字符串数组：

```
String[ ] sensitiveWords1 = {"性","色情","爆炸","恐怖","枪"};//需要屏蔽的敏感词
String[ ] sensitiveWords2 = {"暴力","拉登","导弹"};       //需要被替换的敏感词
```

其中，sensitiveWords1 中是应该被屏蔽掉的敏感词，当遇到这些词时，不显示出来。而在数组 sensitiveWords2 中的是应该被替换掉的敏感词。这里进行了简单处理，sensitiveWords2 中的敏感词统一用“和平”代替。在实际应用中，敏感词可以来源于数据库或数据文件。

在程序的第一个 for 循环中，遍历 sensitiveWords1 数组中的敏感词，使用 indexOf()方法在聊天内容 chatContent 中一一查找，如果发现了聊天内容中含有敏感词，就调用 replaceAll()方法用空字符串替换掉敏感词。在程序的第二个 for 循环中，使用同样的查找过程，只是处理方式不同，将含有 sensitiveWords2 数组中的敏感词使用“和平”字符串替换掉。最后输出过滤以后的聊天内容，输出结果如图 5.18 所示。

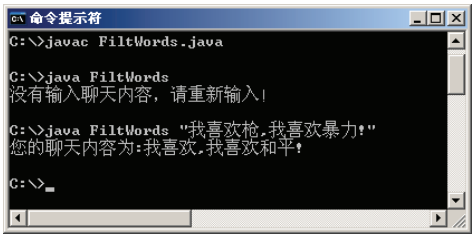


图 5.18　敏感词过滤程序执行结果

5.6　小结

本章深入介绍了 Java 语言中有关数字、字符和字符串的处理的相关知识，重点讲解了 java.lang 包中用来处理数字的 Number 及其子类、专门用来处理数学运算的 Math 类、专门用来处理字符的 Character 类，以及专门用来创建和操作字符串的类 String。

对字符串的处理包括字符串的连接、比较、操纵、获取长度、字符串分割、字符及字符串的替换等。字符串的处理在任何编程语言当中都是非常重要的，也是读者学习本章知识的重点和难点。从第 6 章开始，将向读者介绍 Java 语言中最重要的类和对象的概念。

2

第 2 篇
Java 面向对象篇

- 第 6 章　类和对象
- 第 7 章　继承
- 第 8 章　接口
- 第 9 章　抽象类与嵌套类
- 第 10 章　泛型
- 第 11 章　包
- 第 12 章　集合

第 6 章　类和对象

面向对象是一种思想，最初起源于 20 世纪 60 年代中期的仿真程序设计语言 Simula I。面向对象思想将客观世界中的事物描述为对象，并通过抽象思维方法将需要解决的实际问题分解成人们易于理解的对象模型，然后通过这些对象模型来构建应用程序的功能。通过使用面向对象这种开发风格，程序员能够创建更加可靠、更加容易理解并且更具有复用性的程序。



6.1　面向对象简介

面向对象编程也叫做 OOP（Object Oriented Programming），它是一种创建计算机程序的方法，是新一代的程序开发模式。它模拟显示世界的事物，把软件系统抽象成各种对象的集合，以对象为最小系统单位，这更接近于人类的自然思维，给程序开发人员更灵活的思维空间，使得开发大型复杂业务程序所用的代码量更小，并且成本更低、效率更高而且运行得更加健壮。

6.1.1　什么是对象

在面向对象编程模式出现之前，开发程序传统采用过程化的编程方式，即将程序考虑成一个顺序执行的指令串。所以采用过程化开发方式的程序员首先要做的事，就是针对需求，自顶向下、逐步细化，将问题分解成一连串的简单步骤。

但是随着用户需求的不断增加和变更，软件规模越来越大，业务逻辑越来越复杂，传统的这种面向过程的开发方式暴露出来许多问题，如软件开发延迟，开发成本难以控制，工程难以维护等。所以软件工程师们就希望寻求一种新的开发方式。在 20 世纪 80 年代后期，人们提出了面向对象的程序设计方式，简称 OOP。

在面向对象程序设计里，将数据和处理数据的方法紧密地结合在一起，形成类，再将类实例化，形成对象。计算机程序被概念化成协同工作、共同完成所需任务的一组对象集。每一个对象都是程序的一个独立部分，可自行控制并与其他对象进行通信和交互。

对象，是理解面向对象技术的关键。在现实世界中，对象随处可见，如宠物狗，桌子，电视机，自行车，树，鸟，鱼，汽车等。

真实世界对象具有共同的两个特性：它们都有状态和行为。狗有状态（名字，颜色，

品种，饥饿）和行为（叫，刨地，摇尾巴）。自行车也有状态（当前的齿轮档位，当前的脚踏板频率，当前的速度）和行为（变速，变脚踏板速率，刹车）。识别真实世界对象的状态和行为是开始按照面向对象程序设计进行思考的一个很好的方式。

对真实世界的对象，经常需要问两个问题："这个对象可能处于什么状态？"，以及"这个对象可能具有什么样的行为？"。注意到真实世界对象是复杂多变的：桌上的台灯可能只有两个可能的状态（开和关），以及两个可能的行为（开灯，关灯），但是桌面收音机可能有更多的状态（开，关，当前音量，当前频道），以及行为（开，关，调高音量，调低音量，搜寻，扫描，调台）。

可能还注意到有一些对象，它们还包含其他对象，例如，汽车是一个对象，汽车上还有车载音响，这又是一个对象。这些对真实世界的观察全都可以转化为面向对象程序设计的世界。

软件中的对象在概念性上与真实世界中的对象很相似，如图 6.1 所示。它们都包括状态和相关的行为。一个对象在属性（在一些程序设计语言中用变量代表）中存储它的状态，并通过方法（在一些程序设计语言中是函数）暴露它的行为。也就是说：类将数据封装隐藏在自己内部了，访问这些数据属性必须通过类公开的方法，或者接口。

隐藏内部状态并要求所有的操作通过对象的方法来执行，就是众所周知的数据封装——一个面向对象程序设计的最重要的特点。

正如现实世界中的电视机，用户可以使用遥控器打开或关闭电视，切换频道，调高或者调低音量大小，这些对电视机的操作，只能通过遥控器上暴露给用户的按钮（公开的操作方法）来操作，至于电视机内部是如何实现这些功能的，对用户来说不需要了解。这就是典型的数据封装。例如，我们考虑一下一台电视机，如图 6.2 所示。

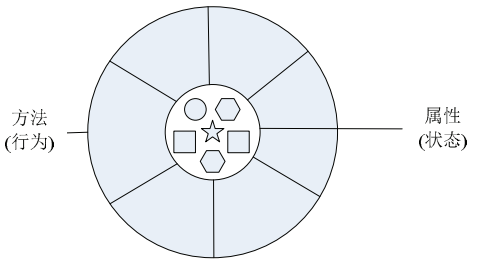


图 6.1　一个软件对象

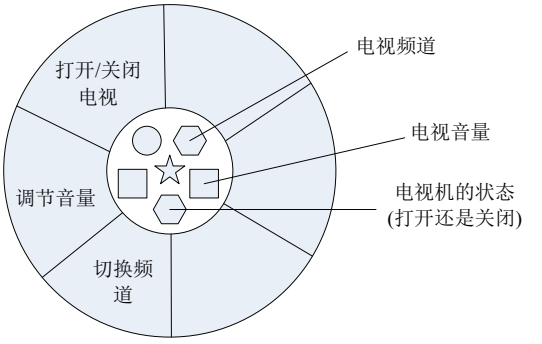


图 6.2　一个作为软件对象的电视机模型

通过属性状态（如当前所在的频道，当前的音量大小，和当前的状态是打开还是关闭）并提供改变这些状态的方法（切换频道，调节音量，打开或关闭电视），对象保持对外部世界的通信，并允许外部如何来对其自身进行操作控制。例如，如果电视机只有 50 个频道，那么切换频道的方法可以拒绝任何小于 1 或大于 50 的值。

将代码封装进各自独立的软件对象中，有一系列的优点，包括：

- 模块化。一个对象的源代码可以被独立于其他对象的源代码进行编写和维护。一旦创建了一个对象，这个对象就能在系统当中被很容易地传递。
- 信息隐藏。只能通过对象提供的方法与其进行交互，而对象的内部实现的细节保

持对外部世界的隐藏。

- 代码重用。如果一个对象已经存在（它也许是另外一个程序员编写的），那么程序员也可以在自己的程序中使用那个对象。这允许在软件开发过程中合理地分工合作，如指定特定的专家来实现/测试/调试复杂的、特定任务的对象，而程序员可以在自己的代码中信任并使用它们。
- 可插件性和更容易调试：如果一个特定的对象产生问题了，程序员可以简单地从他的应用程序中移除这个有问题的对象，并插入另一个对象作为它的替代。这与在现实世界中修理机器是相似的。如果一个螺钉断裂了，就替换它，而不是替换整个机器。

6.1.2 什么是类

在现实世界中，会经常发现许多单独的对象都是同一类型。例如，也许有成千上万的单个的自行车存在，它们都是同样的模型，以相同的方式被制造。每一个自行车都是构建于同一模板并因此包含同样的组件。在面向对象术语中，就可以说，某个人的具体的自行车是自行车这个对象类的一个实例。一个类就是一个模板，从它那里个体对象被创建。

再例如，在课堂上，一个老师对学生提问，他应该这样说：“张小明同学，请回答这个问题”。在这句话中，有两个对象，其中一个对象是一个具体的学生张小明；另一个对象是一个特定的问题，即这个问题。

如果老师这样提问：“同学，请回答问题”，那么会发生什么情况呢？显而易见，这里没有指明具体是哪个同学回答什么问题。在第二个提问中，“同学”是学生的统称，而张小明是学生当中的具体一员，用面向对象的术语来描述的话，“同学”就是一个类，而“张小明”是“同学”中的一个具体的个体，是“同学”类的一个实例，一个具体对象。在程序中，要操作一个对象，必须使用类的一个实例，既类的对象。

在现实生活当中，这样的例子多不胜数。例如一个小孩子看见一个汽车，可能会惊叹：“哇，这辆汽车真是太漂亮了!”，在这里，他不是泛指所有的汽车，而是特定的某一个汽车对象。

所以，在面向对象的程序设计语言中，类代表一个对象类型，它就像一个模板一样，在代码运行阶段它将被创建为一个个对象实例。每个类由两个部分组成：属性和行为，属性通常是一些数据状态值，也就是说：类将数据封装隐藏在自己内部了，访问这些数据属性必须通过类公开的方法，或者接口。

例如自行车类，都有型号、尺寸、品牌，用来描述自行车的特征，有前进、刹车等，用来描述行为。在这里，型号、尺寸、品牌都是自行车类的属性，而前进、刹车等，就是自行车类的方法。自行车类还可以提供如“查看说明书”方法，用户只能通过调用“查看说明书”来了解自行车的型号、尺寸、品牌等特征。

再例如学生类，凡是学生，都有学号、姓名、班级等，这些都是描述学生特征的，称为学生类的属性；凡是学生，都有回答问题、做实验等行为，称为学生类的方法。学生类还可以提供“查看学生档案”方法，其他人可以通过调用“查看学生档案”这个方法来了解学生的基本信息。

也就是说，不管是自行车类，还是学生类，都将自身的信息（属性）封装隐藏在自己内部，但是对外提供了相应的访问方法。外界想要得知这些对象的信息，必须通过调用公

开对外的方法来获得这些信息。这就是面向对象的很重要的特征封装。

别小看这样一个小小的封装，却决定了以后代码的维护性和扩展性。打个比喻，日常生活中经常用各种盒子和袋子包装一些东西，这样做就是为了方便这些东西的携带或储藏。小到生活，大到客观世界每个地方，都是包装分类的影子。例如，公司无论大小都是一个封装，行政部分单位划分也是一种封装，仓库物流更需要封装。可见封装对于人类的生活多么地重要，同样对于程序设计也是非常重要的。

6.1.3　什么是继承

不同类型的对象，相互之间经常有一定数量的共同点。例如，吉普车、小轿车、卡车，都共享汽车的特性（品牌、排气量、当前速度等）。同时，每一个对象还定义了额外的特性使得它们与众不同。例如吉普车的底盘高度；小轿车的载人量；而卡车有额外的载重量。

再例如，小学生、中学生、大学生都是学生，都共享学生的特性（学号、姓名、班级等）。同时，每一个还定义了额外的、自己独有的而别人没有的特性，如小学生有一个是否少先队员的属性；中学生有一个参加高考的行为；而大学生有专业属性，是小学生和中学生所没有的。

面向对象程序设计允许类从其他类继承通用的状态和行为。在图 6.3 中，汽车类现在成为了吉普车、小轿车、卡车的父类。

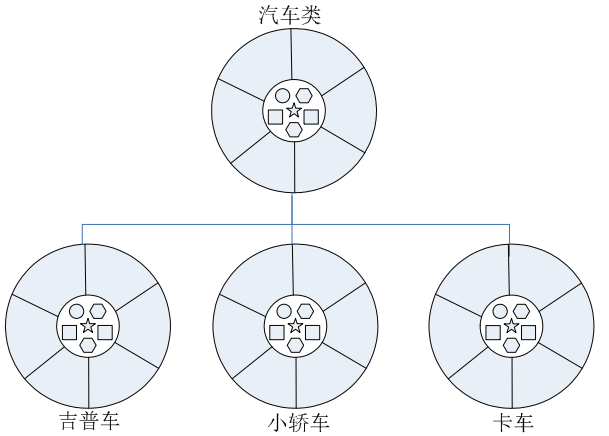


图 6.3　汽车类的一个继承层次

在 Java 程序设计语言中，每一个类被允许有一个直接的父类，每一个父类有可能有无限多的子类。使用继承，可以快速地创建新的类。

6.1.4　什么是接口

正如已经学习过的，对象通过它们暴露出来的方法定义它们与外部世界的交互。方法形成对象与外部世界的接口；例如，电视机前面的按钮，DVD 机的按钮，饮水机的按钮等。按下“电源”按钮，打开或关闭电视机、DVD 机或饮水机。

对于用户来讲，不必关心它们内部是怎么实现的。它们内部实现打开或关闭的原理可能不相同，但对于用户来说，它们的操作是相同的，只需要“按下”按钮，就可以打开或关闭。所以按钮就是一个电视机、DVD 机或饮水机对外部的接口。

在 Java 中，接口的更通常的形式，是一组带有空内容体的相关方法。实现一个接口，允许一个类所提供的行为更加统一。如上述的电视机、DVD 机和饮水机，它们都实现了相同的“按钮”接口，这样对外界来说它们的行为是统一的：只需用户操作按钮，就可以打开相应的电器。至于其内部的复杂实现，用户不需要知道。

接口在类和外部世界之间形成了一个契约的关系，并且在编译器编译阶段这种契约是强制性的。如果一个类声明实现一个接口，那么在类能被成功地编译之前，在接口中定义的所有方法必须出现在类的源代码中。

6.1.5 什么是包

一个包是组织一系列相关类和接口的一个命名空间。从概念上理解，可以将包想象为与计算机上的不同的文件夹相似。在计算机中，也许会将 HTML 页面放到一个文件夹中，图像放在另一个文件夹中，脚本或应用程序再放到另一个文件夹中。由于用 Java 程序设计语言编写的软件可以由成百上千个单独的类组成，很显而易见，通过将相关的类和接口放到包中会使开发工作变得更有条理。

Java 平台提供一个规模庞大的类库（一系列的包）用于在程序员编写的应用程序中使用。这个类库就是有名的“应用程序接口”，或简称“API”。它里面的包代表了与通用程序相关的最常用的任务。

例如，一个 String 对象包含字符串的状态和行为；一个 File 对象允许一个程序员来轻松地创建、删除、检查、比较或修改文件系统中的一个文件；一个 Socket 对象允许用来创建和使用网络 sockets；各种各样的 GUI 对象控制按钮和复选框，以及任何其他的与图形用户接口相关的事物。不夸张地说，有数千个类可供选择使用。这就允许程序员将精力集中于特定应用程序的设计上，而不是集中在基础架构上。

Java 平台 API 规范包含有 Java 平台 6 标准版所提供的全部的包、接口、类、字段和方法的完整列表。作为一个程序员，它将会成为唯一最重要的参考文档。

 6.2 定义类

前面讲了，类是用来创建对象的模板，包含被创建对象的属性和方法的定义。因此，要学习 Java 编程就必须学会怎样去编写类，即怎样用 Java 的语法去描述一类事物共有的属性和行为。

在 Java 中，对象的属性通过变量来表示，即类的成员变量，而对象的行为通过方法来实现，即类的成员方法。方法可以操作属性形成一定的算法来实现一个具体的功能。类把属性及对属性进行操作的相关方法封装为一个整体。

6.2.1 类的基本结构

在 Java 语言中，类是程序构成的基本要素，程序是由类组成的。使用面向对象的语言开发程序，就犹如使用一个个零件组装机器一样，大大降低了开发的难度，提高了开发的效率。因此，熟练掌握类及类的使用，是衡量一个程序员水平的重要标志。

Java SE 6 自带有庞大的类库（API）以供程序员使用，但是必须了解和熟悉类的结构



才能熟练地使用它们。同时，还有很多工作是 Java 的类库所不能涵盖的，需要程序员自己开发适合的类来使用。一个 Java 的类主要包括两个部分：类的声明和类的主体（类体）。

1．类的声明

最基本和最简单的类的声明，包括类的名称，类名前面加关键字 class，类名后面紧跟一对花括号。类的声明如以下代码所示。

```
class 类的名称{
    //属性、构造函数和方法声明
}
```

其中 class 关键字是必需的。类的名称用来标识一个类，它的命名要符合 Java 标识符命名规范。通常类的名称的第一个字母要大写，例如：

```
class AutoCar{
    //属性、构造函数和方法声明
}
```

上面的代码声明了一个汽车类，名称由两个单词组成，每个单词的首字母要大写。上面的类的声明是很“迷你”的——它只包含有类声明所必需的部分。在类的声明中，也可以加上访问权限控制符，以控制对该类的访问，其语法如下所示。

```
[类修饰符] class 类的名称{
    //属性、构造函数和方法声明
}
```

在这里，用方括号将类修饰符包围起来，它代表的含义是里面的内容是可选的。类修饰符可以是 public，private，abstract，final 和 strictft。这里仅介绍与访问权限相关的修饰符 public 和 private，其他三个在后面相关部分再详细讲解。public 和 private 也是 Java 中的关键字，它们决定其他类能否访问本类或能访问什么。public 用在类的声明前面，表示定义的类可以被 Java 的所有软件包使用。如果在类前面不含 public 关键字，则定义的类只能在当前的软件包中使用。例如，可以这样定义类 AutoCar。

```
public class AutoCar{
    //属性、构造函数和方法声明
}
```

在同一个 Java 源文件中可以有多个类，但不能包含两个或两个以上的带有 public 修饰词的类。如果某个类的声明前含有 public 修饰词，那么含有此类的源文件必须与此类的类名相同。例如一个 Java 源文件中含有上述声明 AutoCar 类的代码，那么这个源文件必须命名为 AutoCar.java。

在声明一个类时，还可以提供更多的信息。例如，指明该类的父类（也称为超类）的名称，即该类是从哪个类派生过来的，这需要在类名后面加上关键字 extends 来实现，例如：

```
[类修饰符] class 类的名称 [extends 父类名称]{
    //属性、构造函数和方法声明
}
```

这里的中括号的意思是其内容是可选的。还可以在声明一个类时指明该类是否实现接口，这需要在类名后面加上关键字 implements 来实现，例如：

```
[类修饰符] class 类的名称 [extends 父类名称][implements 接口名称]{
```

```
    //属性、构造函数和方法声明
}
```

关于继承和接口，将在后面的章节讲解。

2．类的主体

类的主体简称类体，指的是类名后面大括号中的内容。类体包含所有用于创建自该类的对象的生命周期的代码，包括构造器（用于初始化新对象）、属性声明（用于表示类及其对象的状态）及方法（实现类及其对象的行为）。一般情况下，类的声明可以按顺序包括以下这些内容：

- 修饰符，如 public，private 及许多其他修饰符。
- 类名，并且按惯例首字母要大写。
- 父类（超类）的名称，都要在前面加上关键字 extends。一个类只能继承自一个父类。
- 被该类实现的接口列表（用逗号进行分隔），在接口前面加上关键字 implements。一个类可以实现多个接口。
- 类体，用花括号{}包围。

6.2.2 声明类的成员变量

在程序中有多种类型的变量，如字段、局部变量、参数等。这些变量在程序中的位置不同，所起的作用也不相同。下面对这些变量加以说明。

- 在一个类中的成员变量——被称为字段。
- 在一个方法中或代码块中的变量——被称为局部变量。
- 在方法声明中的变量——被称为参数。

其中，字段的声明由三部分组成，其语法格式如下。

```
[修饰符] 数据类型  变量名称;
```

组成字段的三部分按顺序依次为：

- 0 个或多个修饰符，如 public 或 private；
- 字段的类型；
- 字段的名称。

例如，在上一节声明的 AutoCar 类，它有三个属性：品牌、排气量和速度。可以使用下面的代码来定义三个字段来代表 AutoCar 类的这三个属性。

```
public String brand;                    //代表品牌
public float exhaust;                   //代表排气量
public int speed;                       //代表速度
```

AutoCar 类的字段被命名为 brand，exhaust 和 speed。关键字 public 说明这些字段是公共成员，可以被任何能够访问该类的对象所访问。下面对构成类的成员变量声明的每一个部分详加说明。

1．访问修饰符

成员字段最左边的修饰符可以让程序员控制其他类对成员字段有什么样的使用权限。目前只考虑 public 和 private（其他访问修饰符将在后面的章节讨论）。



- public 修饰符：可以从所有的类访问该字段。
- private 修饰符：只能从它所在的类内部访问该字段。

根据封装的原则，通常使用私有字段（即使用 private 修饰符来定义字段），这意味着这些字段只能从类的内部被直接访问。如下面定义的 AutoCar 类。

```
public class AutoCar{
     private String brand;                    //表示汽车的品牌
     private float exhaust;                   //表示汽车的排气量
     private int speed;                       //表示汽车的速度
}
```

在这里，声明了三个 private（私有的）字段，没有赋初值。在 Java 中规定，如果声明类的成员变量没有初始化，那么 Java 虚拟机会赋给其默认的初始值：String 类型的为 null，float 类型的为 0.0，int 类型的为 0，boolean 类型的为 false。

2．数据类型

所有的成员变量都必须有一个类型。可以是原始数据类型，如 int、float、boolean 等。也可以是引用类型，如字符串、数组或对象。

3．变量名称

所有的变量，不管是字段，还是局部变量或者参数，都遵循 Java 中关于标识符的命名规范。方法名和类名也遵循同样的命名规范，但在以下方面与变量命名不同。

- 类名的首字母应该大写。
- 方法名的第一个（或者唯一一个）单词应该是一个动词。

6.2.3　声明类的成员方法

所谓方法，类似于其他语言中的函数，是一个相对独立的代码语句块，通常实现一定的功能。在类中声明成员方法的语法格式如下。

```
[方法修饰词列表] 返回类型 方法名称(方法的参数列表){
    //方法体语句
}
```

说明：

- 其中用方括号括起来的部分为可选项，即可以不含该选项。
- 格式第一行的内容称为成员方法定义的头部，或者称为成员方法的声明。
- 在上面的格式中，返回类型指定当前这个方法执行以后，需要返回数据的数据类型，可以是原始数据类型，如 int、float、boolean、char 等，也可以是引用数据类型，如数组名、类名。如果该方法执行以后不返回任何数据，则此处应写上关键字 void，否则将出现编译错误。
- 方法名要符合 Java 标识符命名规范，用来标识当前的成员方法。对该方法的调用需要引用此方法名称。
- 在方法名后面紧跟一对圆括号，里面写上需要传递给该方法的参数。参数也是变量，传进来的参数在方法中可以使用，这是方法接收外部信息的一种方式。参数可以有 0 个、1 个或多个，所以称为参数列表。如果有 0 个参数，圆括号不能省略，里面什么也不写。如果有多个参数，参数之间用逗号分隔。

- 在参数列表后面，紧跟一对花括号，这称为方法体。方法体中是一系列完成特定功能模块的代码。

对于成员方法的声明来说，必须有返回类型、方法名称、一对圆括号，以及在一对花括号之间的方法体。总体来说，方法声明有 6 个部分，按顺序依次如下。

- 修饰符：如 public，private，以及其他一些修饰符（在稍后讲解）。
- 返回类型：方法所返回的数据值的数据类型。或者如果方法不返回值的话，用 void。
- 方法名称：遵循 Java 标识符命名规则，但惯例以动词开头。
- 在圆括号中的参数列表：一个逗号分隔的输入参数列表，前面要加上它们的数据类型。如果没有参数，就使用空的圆括号。
- 一个异常列表，将在后面的章节中讨论。
- 包围在花括号中的方法体：方法的代码，包括局部变量的声明都在这里。

其中，方法的名称和参数类型，组成“方法签名”。例如，可以这样声明一个比较两个整数大小的方法。

```
public int getMaxValue(int value1,int value2){ //方法的签名
     if(value1>=value2)                         //如果 value1 比 value2 的值大
          return value1;                        //方法结束，返回 value1 变量的值
     else                                       //否则
          return value2;                        //方法结束，返回 value2 变量的值
}
```

在上面的方法的声明部分（方法头部），访问修饰符为 public，这说明只要能访问该方法所在的类的对象，都可以调用此方法 getMaxValue()。返回类型为 int，说明这个方法被调用执行以后，需要返回一个整数（在这里返回的是两个整数当中的较大值）。方法名为 getMaxValue，它要符合 Java 标识符命名规则，并且第一个单词为动词。在方法名称后面是一对圆括号，在这里声明了两个参数：value1 和 value2。两个参数都是整数，注意这里声明参数的格式，在每一个参数名前都必须指明该参数的数据类型。参数之间用逗号（,）进行分隔。上述方法声明中的方法签名为：

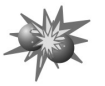
```
getMaxValue(int,int)
```

6.2.4 成员方法命名

虽然方法的名称可以是任何合法的标识符，但是编码惯例上对方法命名有一定的约束。在惯例上，方法名称应该是一个小写的动词，或者一个多词组组成的名称，但是要以一个小写的动词起始，后面跟形容词、名词等。在多词组组成的方法名中，从第二个单词开始，后面的每一个单词的首字母要大写。如下面这些方法的名称所示。

```
run
runFast
getBackGround
getFinalData
compareTo
setX
isEmpty
```

通常，一个方法在一个类中具有唯一的名称。然而，一个方法也可能具有与其他方法相同的名称，这就是“方法重载”。



6.2.5　方法重载

Java 语言支持“重载”方法，并且 Java 能够根据不同的“方法签名”来区分重载的方法。这意味着在一个类中，可以有相同名称但具有不同参数列表的方法（当然会有一些限定条件，这将在“接口和继承”中讨论）。

假设有一个类，它可以绘制各种类型的数据（字符串、整数等），并且它包含一个用来画每一种数据类型的方法。如果为每一个方法都起一个新的名称，是很麻烦的，例如，drawString、drawInteger、drawFloat 等。很明显，这些方法所做的操作基本上都是相似的。

在 Java 语言中，允许程序员为所有这些绘制方法使用相同的名称，但是为每一个方法传递一个不同的参数列表。因此，这个数据绘制类可能会声明 4 个名为 draw 的方法，每一个都有一个不同的参数列表，如下所示。

```
public class DrawDate{
     。。。
     public void draw(String s){            //声明绘制字符串的方法
          。。。
     }
     public void draw(int i){               //声明绘制整数的方法
          。。。
     }
     public void draw(double f){            //声明绘制双精度小数的方法
          。。。
     }
     public void draw(int i , double f){    //声明绘制一个整数和一个小数的方法
          。。。
     }
}
```

通过传递给方法的参数的数量和类型来区分重载的方法。在上面的示例代码中，draw(String s)和 draw(int i)是不同的方法，并且都是唯一的，因为它们要求不同的参数类型。

不能声明超过一个的方法名称相同时参数数量和类型也相同的方法，因为编译器不能区分这些方法。当区分方法时，编译器不考虑返回类型，因此不能声明两个具有相同签名的方法，即使这两个方法有不同的返回类型也不可以。在使用重载的方法时应谨慎，因为这将会降低代码的可读性。

6.2.6　调用成员方法

调用方法，就是执行方法体中的代码语句。在程序中，要调用方法，必须指明调用哪个对象的方法，因为在面向对象的语言中，方法是封装在对象中的，就好比让一个学生唱歌，必须指明让“谁”来唱歌。这里“谁”就是对象，唱歌就是这个对象的方法。如果不指明的话，只说让“唱歌”，没有人知道是让谁来唱歌。

当调用某个对象的方法时，程序流程就会转向方法定义中的第一条语句，并顺序执行其中的代码，直到遇到 return 语句或右花括号为止。

【例 6.1】类之间消息传递（方法调用）示例。

```
public class PassMessageDemo{
```

```
    public static void main(String[] args){
        Hello hello = new Hello( );          //声明一个 Hello 类的实例对象
        hello.sayHello( );                   //调用对象 hello 的 sayHello 方法
        hello.sayHello( );                   //调用对象 hello 的 sayHello 方法
        hello.sayHello( );                   //调用对象 hello 的 sayHello 方法
    }
}
class Hello{
    //定义一个表示问候的成员方法
    void sayHello( ){
        System.out.println("Hello,good morning!");
    }
}
```

程序输出结果如下。

```
"Hello,good morning!"
"Hello,good morning!"
"Hello,good morning!"
```

上面的程序中，重复输出相同的问候。关于调用对象的方法，在后面的章节将给出详细的解释。可以看出，对于重复性的操作，使用方法将会使代码更加简洁。

在程序中，定义成员方法 sayHello()时，因其仅仅是简单的输出，不需要返回值，所以返回类型为 void。而且此方法不需要参数，因此参数列表为空，但圆括号不能省略。当调用 sayHello()方法时，只需要在对象名后面使用圆点运算符（.）引用其方法名即可。在调用方法时，主程序暂停，转而执行 sayHello()方法中的代码，直到执行完毕，再转回主程序继续执行后面的语句。

接下来继续完成上一节的 AutoCar 类的定义。在上一节定义的 AutoCar 类中，声明了 private（私有的）类型的成员变量（字段），这样就不能从类的外部访问到这些变量的值。然而有时需要使用这些值，那么当访问这些字段时，就需要通过间接的方法，即添加获取字段值的公共方法来实现。

【例 6.2】具有 get 和 set 方法的 AutoCar 类。

```
public class AutoCar{
    private String brand;                        //表示汽车的品牌
    private float exhaust;                       //表示汽车的排气量
    private int speed;                           //表示汽车的速度
    public String getBrand(){                    //获得属性 brand 的值
        return brand;
    }
    public float getExhaust(){                   //获得属性 exhaust 的值
        return exhaust;
    }
    public int getSpeed(){                       //获得属性 speed 的值
        return speed;
    }
    public void setSpeed(int newSpeed){          //设置属性 speed 的值
        speed = newSpeed;
    }
}
```

在上面的程序中，因为汽车的品牌和排气量是不允许用户私自改动的，所以只定义了 get 方法，即只能获取这两个属性值，但是不能改变它们的值，这样的方法称为“只读方



法”。而对于车速，肯定是允许用户改变的，所以既有 get 方法用来获取当前车速，也有 set 方法来改变车速（通过传递参数实现），这样的方法称为“读写方法”。

如果要让上述程序运行起来，需要添加一个启动程序类，即含有 main 方法的类，如下面的示例程序所示。

【例 6.3】使用 AutoCar 类示例。

```
class AutoCar{
     private String brand;                    //表示汽车的品牌
     private float exhaust;                   //表示汽车的排气量
     private int speed;                       //表示汽车的速度
     public String getBrand(){                //获得属性 brand 的值
          return brand;
     }
     public float getExhaust(){               //获得属性 exhaust 的值
          return exhaust;
     }
     public int getSpeed(){                   //获得属性 speed 的值
          return speed;
     }
     public void setSpeed(int newSpeed){      //设置属性 speed 的值
          speed = newSpeed;
     }
}
public class Test{
      public static void main(String[] args){
           AutoCar autoCar = new AutoCar( ); //创建一个 AutoCar 类的实例对象
           System.out.println(“当前汽车的品牌为:” + autoCar.getBrand());
           System.out.println(“当前汽车的排气量为:” + autoCar.getExhaust());
           System.out.println(“当前汽车的车速为:” + autoCar.getSpeed());
      }
}
```

注意这里将类 AutoCar 声明中的关键字 public 去掉了，因为在一个程序中只能有一个类为 public，并且该类名与文件名相同。将上面的程序保存为 Test.java，然后进行编译、运行，运行结果如图 6.4 所示。

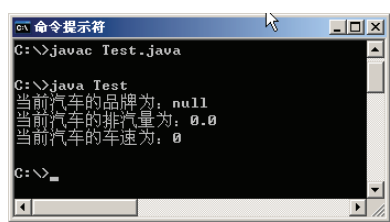


图 6.4　输出 AutoCar 实例的属性值

6.2.7　为类提供构造器

类包含有构造器，当创建对象实例时，构造器会被调用以根据类模板创建对象实例。构造器的声明看上去与方法声明类似，但是构造器有自己的特点。

- 构造器的名称必须与类名相同。
- 构造器没有返回类型，包括关键字 void 也不能有。
- 任何类都含有构造器。如果没有显式地定义类的构造器，则系统会为该类提供一个默认的构造器。这个默认的构造器不含任何参数。一旦在类中显式地定义了构造器，系统就不会再为这个类提供默认的构造器了。

例如，上一节讲的汽车类，可以显式地定义一个构造器，如下面的代码所示。

```
public AutoCar(String originalBrand,float originalExhaust,int startSpeed){
```

```
    brand = originalBrand;     //用参数 originalBrand 的值初始化字段 brand
    exhaust = originalExhaust; //用参数 originalExhaust 的值初始化字段 exhaust
    speed = startSpeed;        //用参数 startSpeed 的值初始化字段 speed
}
```

通常通过在构造器中传入参数，对字段进行初始化，以达到初始化所创建的对象实例的目的，如上述代码所示。在创建一个新的 AutoCar 类的对象实例 autoCar 时，通过 new 运算符调用构造器来实现。

```
AutoCar autoCar = new AutoCar(“奔驰”,1.5f,80);
```

语句“new Auto Car(“奔驰”,1.5f,80);”在内存中为对象 autoCar 分配空间，并初始化它的字段。一个类可以有多个构造器，包括不带参数的构造器。例如下面的这个构造器为不带参数的构造器。

```
public AutoCar(){
    brand = ”奔驰” ;
    exhaust = 1.5f;
    speed = 80;
}
```

如果使用这个不带参数的构造器来创建新的对象，可以使用以下代码：

```
AutoCar newAutoCar = new AutoCar();
```

两个构造器都可以在 AutoCar 中声明，因为它们具有不同的参数列表。与重载方法类似，Java 基于参数列表中的参数数量及其类型来区分构造器。不能为同一个类写两个具有相同数量和类型参数的构造器，否则将会引起编译时错误。

也可以不提供构造器，但是这时候一定要小心。Java 编译器自动地为任何没有构造器的类提供一个无参的默认构造器。这个默认的构造器将调用其超类的无参构造器。在这种情况下，如果超类没有一个无参的构造器，编译器将会报错，因此程序员必须对此进行验证。如果一个类没有显式的超类，那么它隐含的超类是 Object，Object 类有一个无参的构造器。

也可以在一个构造器声明前使用访问控制修饰符，以控制哪一个类可以调用此构造器。如果另外一个类不能调用这个类的构造器，那么它就无法创建这个类的对象。

 6.3 向方法或构造器内传递信息

在方法或构造器的声明中，声明了参数的数量和类型。通过参数，可以向方法或构造器内传递信息。在创建对象时，通过构造器的参数来初始化对象的初始状态；在对象间传递消息时，是通过调用方法并将消息封装在参数中实现的。本节就介绍如何向方法或构造器内传递信息。

6.3.1 使用参数

当需要向一个对象传递消息时，可以通过调用对象的方法并传递参数来实现。例如，下面是一个计算家庭贷款每月还款额的方法，基于贷款额、利率、贷款期限（还款周期数），以及贷款的最终值：

```
public double computePayment(double loanAmt,            //代表贷款额
```



```
                        double rate,                    //代表贷款利率
                        double futureValue,             //代表贷款的最终值
                        int numPeriods) {               //代表还款周期
    double interest = rate / 100.0;
    double partial1 = Math.pow((1 + interest), -numPeriods);
    double denominator = (1 - partial1) / interest;
    double answer = (-loanAmt / denominator)
                - ((futureValue * partial1) / denominator);
    return answer;
}
```

上面的方法中计算贷款的方式有些复杂，暂时不去管它。看一看这个方法的参数，总共有 4 个：贷款额（loanAmt）、利率（rate）、贷款最终值（futureValue），以及还款周期数。前三个参数都是双精度浮点数，第四个参数是一个整数。参数被用在方法体中，并且在运行时接收传递进来的实参数值（这里的参数指的是形参，是方法声明中的参数变量列表。而实参是在方法被调用时实际传递进来的值。当调用一个方法时，实参和形参必须在数据类型和顺序上相匹配）。

6.3.2　实参与形参的关系

在调用对象的方法时，实际向方法传递的参数称为“实参”；而在定义方法时，声明的方法参数称为“形参”。很多初学者在对实参和形参的理解上常常感到迷惑。实际上，要理解实参和形参的不同，一定要从内存变化的角度上去看，就很容易理解了。下面通过例 6.4 来介绍实参与形参的关系，以及它们在内存中的变化。

【例 6.4】编写一个类，定义一个成员方法 getMaxValue，它返回两个参数中较大的值。

```
public class ParameterDemo{
    public static void main(String[] args){
         int a = 5,b=8;           //声明变量 a 和 b，并赋初值
         int max;                 //声明变量 max
         max  = getMaxValue(a,b); //调用 getMaxValue()方法，并将返回值赋予 max
         System.out.println(“最大值是”+max);//输出 max 的值
    }
    //定义对两个整数进行比较的成员方法
    public static int getMaxValue(int value1,int value2){
        if(value1>=value2)                  //比较形参 value1 和 value2 的大小
            return value1;                  //如果 value1 大，返回其值
        else                                //否则
            return value2;                  //返回 value2 的值
    }
}
```

上面的程序的输出结果如下。

```
最大值是 8
```

在上面的程序中，变量 a 和 b 称为“实参”，而方法 getMaxValue 后面参数列表中的参数 value1 和 value2 称为“形参”。关于形参和实参在内存中的变化，如图 6.5～图 6.7 所示。

在上面的程序中，当在 main 方法中声明变量 a、b、max 时，程序在内存中为这三个变量分配空间，其中变量 a 和 b 分别赋予初值 5 和 8，而 max 没有明确赋初值，Java 会自动地为其赋予初值 0，如图 6.5 所示。

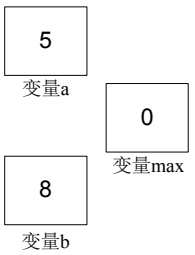


图 6.5　实参内存情况

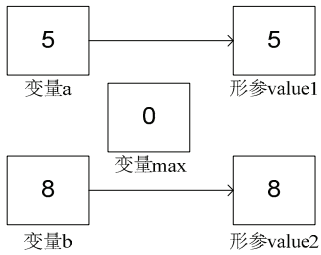


图 6.6　实参传值给形参

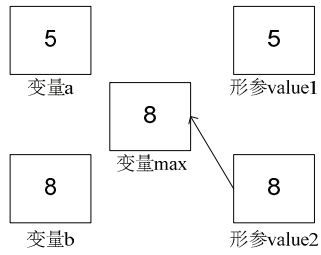


图 6.7　方法调用返回值赋给变量 max

当调用 getMaxValue()方法时，程序转向调用成员方法 getMaxValue()，并且在内存中为形参 value1 和 value2 分配内存，同时将实参值按顺序一一赋予形参，这里将 a 的值赋予 value1，将 b 的值赋予 value2。因此 value1 的值等于 5，value2 的值等于 8，如图 6.6 所示。这里实参与形参的关系仅仅是传递值给形参，此后再无关系。

在方法 getMaxValue()中，对 value1 和 value2 的值进行比较，返回其最大值，并将返回值赋给变量 max，如图 6.7 所示。此时 max 的值变为返回的最大值 8。然后主程序接着执行后面的语句，输出 max 的值。

一定要注意，当实参和形参均为原始数据类型时，实参与形参的关系仅仅是将实参的值传递给形参，此外实参与形参没有任何关系。所以在成员方法中对形参的改变，不会影响到实参。

【例 6.5】形参与实参关系示例。

```
public class Swap{
     public static void main(String[] args){
          int a = 5,b=8;
          exchangeValues(a,b);                    //调用 exchangeValues()方法
          System.out.println(“交换以后的值为:”);
          System.out.println("a=" + a);
          System.out.println("b=" + b);
     }
     //定义对两个整数进行交换的成员方法，因不需要返回值，在声明中使用 void
     public static void exchangeValues(int value1,int value2){
          int temp = 0;                           //声明一个中间变量
          temp = value1;                          //交换第一步
          value1 = value2;                        //交换第二步
          value2 = temp;                          //交换第三步
     }
}
```

上面的程序的输出结果如下。

```
交换以后的值为:
a=5
b=8
```

可以发现，在方法 exchangeValues 中对两个整数进行了交换，但是对变量 a 和 b 的值并没有影响。程序执行过程中，内存的变化情况如图 6.8～图 6.12 所示。

当程序在 main 方法中声明两个变量 a 和 b 并赋予初值时，其内存如图 6.8 所示。当调用成员方法 exchangeValues()时，实参 a 和 b 将值传递给形参 value1 和 value2，如图 6.9 所示。



图 6.8　实参 a 和 b

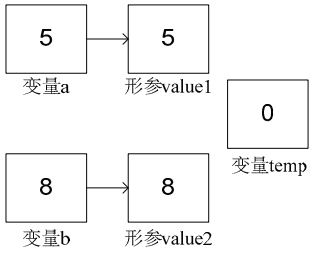


图 6.9　实参传值给形参

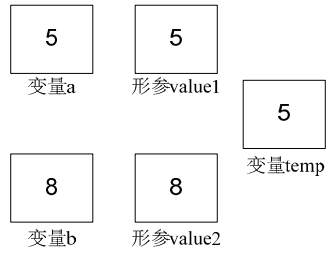


图 6.10　交换第一步

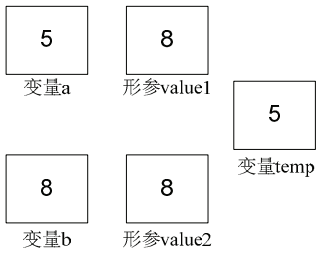


图 6.11　交换第二步

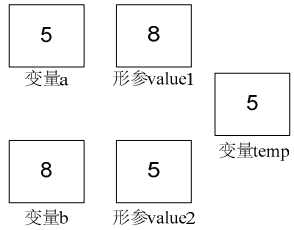


图 6.12　交换第三步

在 exchangeValues()方法内部，开始对 value1 和 value2 的值进行交换。在交换第一步，先将 value1 的值 5 保存在中间变量 temp 中，如图 6.10 所示。然后将 value2 的值 8 赋给 value1，如图 6.11 所示。最后，将中间变量中保存的值 5 赋给 value2，如图 6.12 所示。

方法 exchangeValues()调用结束，只是将形参 value1 和 value2 的值进行了交换，对实参 a 和 b 没有任何影响，因此输出结果 a 和 b 的值保持不变。

6.3.3　参数类型

一个方法或构造器的参数可以是任何数据类型。包括原始数据类型如 double、float、int 等，以及引用数据类型如对象和数组。下面是一个接收一个数组作为参数的方法的示例。在这个示例中，该方法创建一个新的 Polygon 对象并使用一个 Point 对象数组对它进行初始化（这里假定 Point 是一个类，代表一个点的 x、y 坐标）。

```
public Polygon polygonFrom(Point[ ] corners){
     //方法体内容
}
```

需要注意的是，Java 语言不允许将一个方法传递到另一个方法中，但是可以传递一个对象到一个方法中，然后再调用这个对象的方法。

6.3.4　传递任意数量的参数

可以使用“可变参数”来传递任意数量的值到一个方法。当程序员不知道有多少个特定类型的参数要被传递给方法时，使用可变参数。可变参数是手工创建一个数组的简洁方法。

要使用可变参数，在最后一个参数的类型后面跟上省略号“…”，然后是一个空格，再接着是参数名。具有这种可变参数的方法可以接收任意数量的该类型的参数，包括空参数。例如下面代码：

```
public Polygon polygonFrom(Point... corners) {
```

```
        //corners 参数被当作数组处理，可以调用数组属性 length 获得长度
        int numberOfSides = corners.length;
        double squareOfSide1, lengthOfSide1;          //声明两个变量
        squareOfSide1 = (corners[1].x - corners[0].x)*(corners[1].x - corners
[0].x) + (corners[1].y - corners[0].y)*(corners[1].y - corners[0].y) ;
       lengthOfSide1 = Math.sqrt(squareOfSide1); //调用 Math 类的静态方法求平方根
    …
    }
```

可以看出来，在上面这个方法中，corners 被当做数组使用。调用这个方法时，既可以使用一个数组，也可以使用一系列的参数。不管使用哪种形式，方法中的代码都会将参数作为一个数组对待。最经常使用可变参数的地方是打印方法。例如 printf()方法：

```
public PrintStream printf(String format,Object… args)
```

这个方法允许打印任意数量的对象。例如，它可以这样被调用：

```
System.out.printf("%s:%d,%s,%s,%s%n", name,idnum,address,phone,email);
```

6.3.5 参数名称

当在一个方法或一个构造器中声明一个形参时，就提供了一个形参名称。形参的名称在方法体内使用，代表传递进来实参。形参的名称在其作用域内必须是唯一的。不能在同一方法或构造器中有两个相同的参数名，也不能参数名与方法内或构造器内的局部变量的名称相同。

形参的名称可以与类的字段名称相同。在这种情况下，称之为形参“屏蔽”了字段。屏蔽字段会让代码难以阅读，按惯例，它只用于构造器中和设置特定字段的方法中这两种情况。例如，考虑一下下面这个 Circle 类和它的 setOrigin()方法：

```
public class Circle{
     private int x, y, radius;
     public void setOrigin(int x, int y){
          …
    }
}
```

在上面的代码中，Circle 类有三个字段：x，y 和 radius。方法 setOrigin()有两个参数，每一个都与字段名相同。每个方法参数屏蔽与其名称相同的字段。因此当在方法体中简单地使用名称 x 或 y 时，指的是参数 x 或参数 y，而不是字段 x 或字段 y。要访问 x 或 y 字段，必须使用限定名。这在后面的“使用 this 关键字”一节讲解。

6.3.6 传递原始数据类型参数

原始数据类型的参数，如 int 或 double，是通过“值传递”的形式传给方法的，这意味着对参数值的任何改变只影响到方法内部。当方法调用返回时，参数会被释放并且任何对它们所做的改变会丢失。

【例 6.6】编写一个类，定义一个成员方法 passMethod()，向其传递原始数据类型的参数。

```
public class  ValueParDemo {
     public static void main(String[] args) {
         int x = 3;
```



```
        passMethod(x);          //调用方法 passMethod()，使用 x 作为实参
        System.out.println("调用方法 passMethod 以后, x = " + x);
                                //输出 x，看它的值是否被改变了
    }
    public static void passMethod(int p) {
                                // 在方法 passMethod()中改变参数的值
        p = 10;
    }
}
```

编译并运行此程序，输出结果如下。

```
调用方法 passMethod 以后, x = 3
```

可以看到，虽然在方法 passMethod(int)中改变了参数 p 的值。但是因为方法 passMethod()中的参数 p 只是实参的一个拷贝，所以对 p 的值的改变，并不影响实参。

6.3.7　传递引用数据类型参数

引用数据类型参数，如对象，也是通过传值方式传递给方法的。这意味着，当方法返回时，传进来的引用仍然引用先前的对象。然而，对象的字段的值有可能在方法中被改变了（如果有合适的访问级别的话）。例如，考虑下面这个方法 moveCircle()（moveCircle()方法可以定义在任意的类中，用来移动一个 Circle 对象）。

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    //将圆点移动到 x+deltaX, y+deltaY 处
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);
    //将一个新的引用赋给 circle
    circle = new Circle(0, 0);
}
```

假设调用此方法时，传递的参数如下所示。

```
moveCircle(myCircle, 23, 56)
```

在此方法内，形参 circle 首先指向 myCircle 对象。然后分别通过 23 和 56 改变 circle 所引用对象（在这里是 myCircle）的 x 和 y 坐标。当方法返回时，这种改变仍然保持。然后 circle 对象被赋予一个引用，指向一个新的 Circle 对象，其 x 和 y 值均为 0。在方法内部，circle 对象所指向的对象已经改变了，但是当方法返回时，myCircle 仍然引用着方法调用之前的 Circle 对象。

6.4　对象

典型的 Java 程序会创建许多对象，对象间通过调用方法进行交互和通信。通过这些对象的交互作用，一个程序能够执行各种各样的任务，如实现一个 GUI 界面、运行一个动画或发送和接收网络上的信息。一旦一个对象已经完成了它应该完成的工作，它的资源就会被回收以供其他对象使用。

6.4.1　对象实例

下面通过一个简单的程序，说明对象的使用。在下面这个程序 CreateObjectDemo 中，

定义了三个类。其中两个类 Point 和 Rectangle 分别代表点和矩形。在另外一个主程序 CreateObjectDemo 中，创建这两个类的实例对象并使用其属性和方法完成相应的操作。

【例 6.7】创建程序 CreateObjectDemo.java。在这个程序中，创建三个对象，一个 Point 对象和两个 Rectangle 对象。实现步骤如下。

① 打开记事本，输入以下代码：

```
//创建类 Point，代表一个有着 x 坐标和 y 坐标的点
class Point {
    public int x = 0;                  //声明 int 类型变量 x，代表点的水平坐标
    public int y = 0;                  //声明 int 类型变量 y，代表点的垂直坐标
     // 构造器
    public Point(int a, int b) {
          x = a;                       //在构造器中初始化实例变量 a 和 b
          y = b;
    }
}
//创建类 Rectangle，代表一个矩形
class Rectangle {
    public int width = 0;              //声明 int 类型变量 width，代表矩形的宽
    public int height = 0;             //声明 int 类型变量 height，代表矩形的高
    public Point origin;               //声明一个 Point 对象，代表一个点
    // 四个构造器
    public Rectangle() {               //无参构造器
         origin = new Point(0, 0); //默认创建坐标为(0,0)的点
   }
    public Rectangle(Point p) {        //带有一个 Point 类型的参数的构造器
         origin = p;                   //使用已知的点 p 初始化矩形的左上角的点
   }
    public Rectangle(int w, int h) {   //带有两个整型参数的构造器
         origin = new Point(0, 0);     //初始化矩形左上角坐标为(0,0)
         width = w;                    //初始化矩形的宽
         height = h;                   //初始化矩形的高
   }
    public Rectangle(Point p, int w, int h) {  //带有三个参数的构造器
         origin = p;                   //使用已知的 Point 对象初始化矩形左上角坐标
         width = w;                    //使用参数 w 初始化矩形的宽
         height = h;                       //使用参数 h 初始化矩形的高
    }
    // 移动矩形的方法
    public void move(int x, int y) {
         origin.x = x;                 //将矩形左上角点的 x 坐标改变为新的值
         origin.y = y;                 //将矩形左上角点的 y 坐标改变为新的值
    }
    // 计算矩形面积的方法
    public int getArea() {
         return width * height;
    }
}
//主类，含有 main()方法
public class CreateObjectDemo {
    public static void main(String[] args) {
```



```
        //声明并创建一个坐标点对象和两个矩形对象
        Point originOne = new Point(23, 94);  //创建一个 Point 点对象
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
                                    //使用已知的点和长宽值创建一个矩形对象
        Rectangle rectTwo = new Rectangle(50, 100);
                                    //使用默认的坐标和长宽值创建一个矩形对象
      //显示 rectOne 的宽、高和面积
      System.out.println("rectOne 的宽是: " + rectOne.width);
      System.out.println("rectOne 的高是: " + rectOne.height);
      System.out.println("rectOne 的面积是: " + rectOne.getArea());
      //设置 rectTwo 的位置
      rectTwo.origin = originOne;
      //显示 rectTwo 的位置
      System.out.println("rectTwo 的 X 坐标是: " + rectTwo.origin.x);
      System.out.println("rectTwo 的 Y 坐标是: " + rectTwo.origin.y);
      //移动 rectTwo 并显示它的新位置
      rectTwo.move(40, 72);
      System.out.println("rectTwo 的 X 坐标是:" + rectTwo.origin.x);
      System.out.println("rectTwo 的 Y 坐标是:" + rectTwo.origin.y);
   }
}
```

② 将以上程序保存到文件 CreateObjectDemo.java 中，使用如下命令编译并运行。

```
javac CreateObjectDemo.java
java CreateObjectDemo
```

③ 程序输出结果如下。

```
rectOne 的宽是: 100
rectOne 的高是: 200
rectOne 的面积是: 20000
rectTwo 的 X 坐标是: 23
rectTwo 的 Y 坐标是: 94
rectTwo 的 X 坐标是: 40
rectTwo 的 Y 坐标是: 72
```

在上面的这个程序中，创建、操纵、显示了各种对象的信息。在后面的三节中，使用上面的这个例子来描述在一个程序中对象的生命周期。通过这些内容，可以掌握如何在程序中编写创建和使用对象的代码，以及当一个对象的生命周期结束以后，系统是如何进行清理的。

6.4.2　创建对象

正如前面所讲，类是对象的模板，可以从一个类创建一个对象。下面的语句来自于上面所讲的程序 CreateObjectDemo.java，每一个语句都创建一个对象并将其赋予一个变量。

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

第一行创建一个 Point 类的对象，第二行和第三行各自创建一个 Rectangle 类的对象。上述三个语句，每个语句都由三个部分组成。

- 声明：赋值符号（=）左边的代码都是变量声明，声明了三个对象类型的变量名。
- 实例化：关键字 new 用来创建一个新的对象。

➢ 初始化：new 运算符后面跟一个构造器的调用，由构造器来初始化新的对象。

1．声明一个变量指向一个对象

在前面的内容中，学习过如何声明一个变量。声明变量的语法格式如下所示。

数据类型 变量名称;

例如，声明一个整型变量，使用如下代码：

```
int a ;
```

这个声明通知编译器，程序中使用一个变量名“a”指向一个数据类型为 int 型的数据。对于原始数据类型的变量，这样的声明还为该变量分配适当大小的内存。也可以相类似地，声明一个引用类型的变量。例如：

```
Point originOne;
```

如果像上面这样声明 originOne，那么它的值是不确定的，除非真正地创建一个对象并赋给它。简单地这样声明一个引用变量并不会生成一个对象。要真正地生成对象，需要使用 new 运算符。在使用 originOne 之前，必须给它赋一个对象，否则，就会出现编译错误。

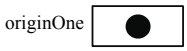


图 6.13　没有引用对象的变量

对于当前没有引用对象的变量来说，可以像图 6.13 所描绘的那样（一个变量名 originOne，再添加一个空指针）。

2．实例化一个类

当使用 new 运算符来实例化一个类时（生成这个类的一个实例对象），它通过为新的对象分配内存并返回一个对该内存的引用来实现。new 运算符还调用对象的构造器。

new 运算符要求后跟一个单一的后缀型参数：对构造器的调用。构造器的名称为要进行实例化的类的名称。new 运算符返回一个对它所创建的对象的引用。这个引用代表其所创建对象的内存地址，通常赋给一个合适类型的变量，例如：

```
Point originOne = new Point(23,86);
```

由 new 运算符所返回的引用不必一定要赋给变量，它也可以直接在一个表达式中使用。例如下面的代码中，直接使用 new 运算符创建一个 Rectangle 对象，并直接调用该对象的 height 属性。不过这样一来，创建的对象不可以重复使用，因为没有保存对它的引用：

```
int height = new Rectangle( ).height;
```

3．初始化一个对象

在本节的开头，讲到了一个 Point 类，它有两个属性 x 和 y。这两个属性分别代表点的水平坐标和垂直坐标。Point 类的定义如下。

```
public class Point {
    public int x = 0;                          //代表点的水平横坐标
    public int y = 0;                          //代表点的垂直纵坐标
    //构造器
    public Point(int a, int b) {               //使用两个参数来初始化 x 和 y 坐标
          x = a;
          y = b;
    }
}
```

这个类只包括一个单一的构造器。在 Point 类中的构造器需要两个参数。下面的代码提供 23 和 86 作为构造器的实参：

```
Point originOne = new Point(23,86);
```

在这行代码中，使用 new 运算符生成一个新的 Point 类的对象，并在内存中为其分配相应的空间，然后将其引用赋给 Point 类型的变量 originOne 保存，执行结果如图 6.14 所示。

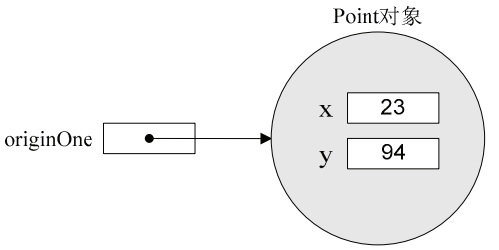


图 6.14　对象 originOne 的内存状态

在本节的开头，还讲到了一个 Rectangle 类，代表一个矩形。其定义如下。

```
public class Rectangle {
    public int width = 0;       //定义一个成员变量，代表矩形的宽，并赋初值为 0
    public int height = 0;      //定义一个成员变量，代表矩形的高，并赋初值为 0
    public Point origin;        //定义一个引用类型的成员变量，代表一个点
    //4 个构造器
    public Rectangle() {
        origin = new Point(0, 0); //默认使用（0,0）来初始化矩形的左上角坐标
    }
    public Rectangle(Point p) {     //带有一个参数的构造器
        origin = p;                 //使用已知的点 p 来初始化矩形的左上角坐标
    }
    public Rectangle(int w, int h) {        //带有两个 int 型参数的构造器
        origin = new Point(0, 0); //默认使用（0,0）来初始化矩形的左上角坐标
        width = w;                  //使用参数 w 初始化矩形的宽
        height = h;                 //使用参数 h 初始化矩形的高
    }
    public Rectangle(Point p, int w, int h) {  //带有三个参数的构造器
        origin = p;                 //使用已知的点 p 来初始化矩形的左上角坐标
        width = w;                  //使用参数 w 初始化矩形的宽
        height = h;                 //使用参数 h 初始化矩形的高
    }
    // 移动矩形的方法
    public void move(int x, int y) {
        origin.x = x;               //将矩形左上角的坐标移动到参数指定的位置
        origin.y = y;
    }
    //计算矩形面积的方法
    public int getArea() {
        return width * height;
    }
}
```

每个构造器都可以使用原始数据类型和引用类型为矩形的面积和宽度提供初始值。如果一个类有多个构造器，这些构造器必须有不同的签名。Java 编译器基于参数的数量和类

型来区分不同的构造器。例如，当 Java 编译器遇到下面的代码时，它就知道需要调用 Rectangle 类中带有一个 Point 参数和两个整型参数的构造器。

```
Rectangle rectOne = new Rectangle(originOne,100,200);
```

执行这行代码，将调用相同签名的构造器，将其中的 origin 成员变量初始化为 originOne。此外，构造器设置 width 字段的值为 100 和 height 字段的值为 200。现在有两个引用指向同一 Point 对象（一个对象可以有多个引用指向它）。执行结果如图 6.15 所示。

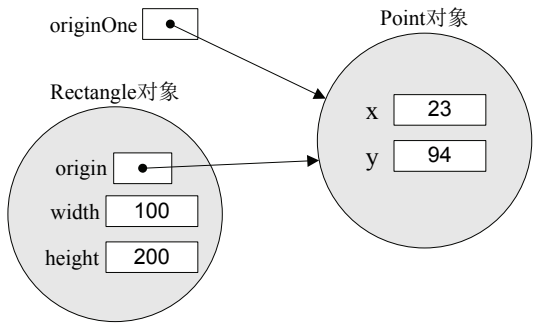


图 6.15　两个引用指向同一个对象

下面这行代码调用带有两个整型参数的 Rectangle 构造器，这两个参数分别代表要创建的矩形的宽和高。

```
Rectangle rectTwo = new Rectangle(50,100);
```

在上面的这个构造器中，为 width 和 height 字段提供了初始值。如果检查一下构造器中的代码，会发现它生成了一个新的 Point 对象，该对象的 x 和 y 值都被初始化为 0。下面这一行代码中使用的 Rectangle 构造器是不带任何参数的构造器，所以它会调用一个无参构造器。

```
Rectangle rect = new Rectangle( );
```

所有的类至少有一个构造器。如果一个类没有显式地声明一个显示器，Java 编译器会自动地提供一个无参构造器，称为“默认构造器”。这个默认构造器会调用它的父类的无参构造器。如果没有显式的父类的话，则调用它的隐含的父类 Object 的构造器。如果它的父类没有构造器（Object 类确定有一个），编译器将会拒绝对该程序进行编译。

6.4.3　使用对象

一旦创建了一个对象以后，就需要在适当的时候使用它。例如需要它的某个字段的值，改变它的某个字段的值，或者调用它的方法来执行一个动作。

1．引用一个对象的字段

通过对象的字段名称来访问对象的字段。在一个类的内部，可以使用简洁的名称来引用字段。例如，可以在上一节中讲到的 Rectangle 类中添加一行输出 width 和 height 值的语句。

```
System.out.println("宽和高分别是: " + width + ", " + height);
```

在这里，width 和 height 是简洁的名称。在对象类的外部的代码，要引用对象中的字段，必须使用一个对象引用或表达式，后面跟一个圆点（.）运算符，再后面跟一个简洁字



段名，例如：

```
objectReference.fieldName
```

例如，在本节开始的 CreateObjectDemo.java 程序中，CreateObjectDemo 类的代码位于 Rectangle 类的外部，因此要调用 Rectangle 的对象 rectOne 中的 origin、width 和 height 字段，在 CreateObjectDemo 类中必须分别使用 rectOne.origin、rectOne.width 和 rectOne.height 来引用，如下面的代码所示。

```
System.out.println("矩形 rectOne 的宽度为: " + rectOne.width);
System.out.println("矩形 rectOne 的高度为: " + rectOne.height);
```

如果在 CreateObjectDemo 类中使用简洁的名称来引用 width 和 height，将会导致编译错误。同一类型的对象对相同的实例字段拥有它们自己的一份拷贝。每一个 Rectangle 对象都有自己的字段 origin、width 和 height。当通过一个对象的引用访问一个实例字段时，引用的是特定对象的字段。例如，在 CreateObjectDemo.java 程序中，两个对象 rectOne 和 rectTwo 都有各自不同的 origin、width 和 height 字段。

除了可以通过对象的引用访问对象的字段之外，也可以使用任何能够返回一个对象引用的表达式。因为 new 运算符返回一个对象的引用，所以可以使用 new 运算符返回的值来访问一个新的对象的字段，如下面的代码所示。

```
int height = new Rectangle( ).height;
```

这个语句生成一个新的对象并马上获取它的高度。实际上，这个语句计算一个 Rectangle 对象的默认高度。注意这条语句执行以后，程序中不再存在一个指向这个生成的对象的引用，因为程序永远没有在任何地方存储此引用。所以这里生成的对象是不可引用的，它所占用的资源会释放并被 Java 虚拟机所回收。

2．调用一个对象的方法

还可以使用一个对象的引用来调用该对象的方法。方法调用的语法格式如下。

```
对象名称.方法名称（参数列表）;
```

或

```
对象名称.方法名称（）;
```

即通过一个对象名后面跟一个圆点运算符，再后面跟上方法名称及参数列表（如果没有参数，也要保留空的圆括号）来调用。在本节开始的 CreateObjectDemo.java 程序中，Rectangle 类有两个方法：getArea()用来计算矩形的面积，move()改变矩形的原点。在 CreateObjectDemo.java 中调用这两个方法的代码如下。

```
System.out.println("rectOne 的面积为: " + rectOne.getArea());
…
rectTwo.move(40, 72);
```

第一行语句调用 rectOne 对象的 getArea()方法并显示结果。第二行移动 rectTwo 到新的原点。和实例字段一样，这里的对象名称必须是对一个对象的引用。可以使用变量名，也可以使用任何能够返回一个对象引用的表达式。因为 new 运算符返回一个对象引用，所以可以使用 new 运算符返回的值来调用一个新的对象的方法，如下面这行代码所示。

```
new Rectangle(100, 50).getArea( );
```

上面的语句中的表达式“new Rectangle(100,50)”返回一个指向 Rectangle 对象的引用，并使用圆点运算符调用新的 Rectangle 对象的 getArea()方法来计算它的面积。

有的方法，如 getArea()，调用后会返回一个值。对于有返回值的方法，可以在表达式中直接调用该方法。可以将返回值赋给一个变量，并使用它来进行判断或者控制循环。下面这行代码将方法 getArea()返回的值赋给变量 areaOfRectangle。

```
int areaOfRectangle = new Rectangle(100,50).getArea( );
```

需要记住的是，在一个特定对象上调用其方法，等同于向该对象发送一个消息。在上面这行示例代码中，被调用 getArea()方法的对象是由构造器返回的 Rectangle 对象。

3．垃圾回收机制

有的面向对象语言要求程序员保持对他自己所创建的对象的跟踪，并且当程序不再需要这些对象时，由程序员显式地销毁它们。这种由程序员来管理内存的方式很明显是非常乏味而且容易出错的。而 Java 平台允许程序员需要多少对象就创建多少（当然是有限制的，这要根据所在系统的处理能力），并且不必关心什么时候销毁这些对象。Java 运行时环境会做出决定，当它认为某些对象不再被使用了，就会删除这些对象。这种处理称为“垃圾收集”。

对于垃圾收集器来说，当一个对象没有引用时，垃圾收集器有选择地进行处理。通常当变量离开它的作用域以后，对这个变量的引用就会被删除。或者，通过将变量设置为 null 来显式地删除一个对象的引用。因为在一个程序中可以有多个引用指向同一个对象，所以只有当所有对该对象的引用都被删除以后，垃圾收集器才会选择处理这个对象。

Java 运行时环境有一个垃圾收集器，它周期性地释放不再被引用的对象的内存。垃圾收集器自动地执行内存清理工作，自主地决定执行的时间。

6.5 方法的返回值

方法实际上是一段单独执行的代码段。当方法执行完毕以后就会返回主程序。不管哪种情况先发生，方法都会返回。当以下情况发生时，方法返回到调用它的代码处：

- 完成方法中所有语句；
- 遇到 return 语句；
- 抛出一个异常（后续章节解释）。

6.5.1 返回值

在方法声明中声明方法的返回类型。在方法体内，使用 return 语句来返回一个值。任何声明 void 的方法都不返回任何值。在声明为 void 的方法中，不需要包含一个 return 语句。当然也可以包含一个 return，在这种情况下，一个 return 语句通常被用于终止一个控制流程块并退出方法，如以下代码所示。

```
return;
```

如果一个方法已经声明为 void，但是在方法内还试图返回一个值的话，编译器将报错。如果一个方法没有声明为 void，那么就必须包含一个 return 语句，后跟相应的返回值。例如：

```
return 返回值;
```



返回值的数据类型必须与方法所声明的返回类型一致。例如，在一个方法声明中返回类型为 int，而方法体内 return 语句返回一个小数，那么将通不过编译。例如：

```
public int getHeight( ){
     …
     return 15.6;
}
```

因为返回值类型与方法声明中的返回类型不一致，这将导致编译错误。在上一节的 Rectangle 类中，声明了一个 getArea()方法，它返回一个整数。

```
public int getArea( ) {
     return width * height;
}
```

这个方法返回表达式 width*height 计算以后的整数值，这是一个原始数据类型。除了可以返回原始数据类型之外，方法还可以返回一个引用类型。例如：

```
public Point getPoint(int x, int y) {
     return new Point(x,y);
}
```

在上面的代码中，返回一个新创建的 Point 对象，该对象使用调用 getPoint(int x,int y)方法时传递的实参值初始化实例对象。返回类型 Point 指明此方法一定要返回一个 Point 对象。

6.5.2　返回类或接口

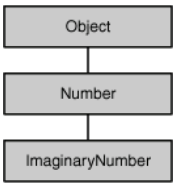


图 6.16　ImaginaryNumber 类的层次结构

当一个方法使用一个类名作为其返回类型的话，如 Point，那么返回对象的类型类必须是返回类型本身，或其子类。假设有一个类层次结构，其中 ImaginaryNumber 类是 java.lang.Number 类的子类，而 Number 类又是 Object 的子类，如图 6.16 所示。

现在声明一个方法，该方法声明返回一个 Number 类型的引用，如以下代码所示。

```
public Number returnANumber() {
     ...
}
```

方法 returnANumber 可以返回一个 ImaginaryNumber，但是不能返回一个 Object，也就是说，该方法可以返回声明的返回类型 Number 的子类，但是不能返回其超类。可以重写一个方法并定义它返回一个子类，如下面代码所示。

```
public ImaginaryNumber returnANumber() {
     ...
}
```

这个技术，称为“协变返回类型”，它的意思是在返回类型上允许在子类方向上进行变化（这是在 JDK1.5 中加入的新特性，即在面向对象的编程中，子类在重写父类的方法时可以改变这个方法的返回值的类型，但前提是子类的返回值是父类那个返回值的子类）。还可以使用接口名称作为返回类型。在这种情况下，返回的对象必须实现指定的接口。

6.6　使用 this 关键字

在一个实例方法或一个构造器中，关键字 this 是对当前对象的引用。所谓当前对象，

指的是其方法或构造器正在被调用的对象，也就是正在调用的方法或构造器所在的对象。可以通过 this 在一个实例方法或构造器中引用当前对象的任何成员。

6.6.1 对字段使用 this

最经常使用 this 关键字的情况，是在对象的一个字段被方法或构造器的参数屏蔽时，需要调用这个被屏蔽的字段的这种情况。例如，像下面的代码一样定义 Point 类。

```
public class Point {
    public int  x = 0;              //声明公共字段 x
    public int  y = 0;              //声明公共字段 y
    //构造器
    public Point(int a, int b) {
         x = a;                     //初始化字段 x 的值
         y = b;                     //初始化字段 y 的值
    }
}
```

也可以像下面这样来写。

```
public class Point {
    public int  x = 0;              //声明公共字段 x
    public int  y = 0;              //声明公共字段 y
    //构造器
    public Point(int x, int y) {
         this.x = x;                //初始化字段 x 的值，this 代表当前的对象实例
         this.y = y;                //初始化字段 y 的值，this 代表当前的对象实例
    }
}
```

在第二个构造器中，每一个参数 x 和 y 都屏蔽了对象的字段 x 和 y。在构造器中，x 是构造器的第一个参数的一个本地副本。要引用 Point 的字段 x，构造器必须使用 this.x。

6.6.2 对构造器使用 this

在一个构造器中，还可以使用 this 关键字来调用同一个类中的另外的构造器，这种做法称为“显式构造器调用”。下面是另外一种实现形式的 Rectangle 类。

```
public class Rectangle {
    private int x, y;           //声明两个私有类型的成员变量，代表矩形左上角的点
    private int width, height; //声明两个私有类型的成员变量，代表矩形的宽和高
    public Rectangle() {        //默认构造器
        this(0, 0, 0, 0);       //调用当前对象实例的带有 4 个参数的构造器
    }
    public Rectangle(int width, int height) {  //带有两个参数的构造器
        this(0, 0, width, height); //调用当前对象实例的带有 4 个参数的构造器
    }
    public Rectangle(int x, int y, int width, int height) {
                                   //带有四个参数的构造器
        this.x = x;                //初始化当前对象实例的字段 x
        this.y = y;                //初始化当前对象实例的字段 y
        this.width = width;        //初始化当前对象实例的字段 width
        this.height = height;      //初始化当前对象实例的字段 height
```



```
    }
    ...
}
```

这个类包含一系列的构造器。每个构造器或者初始化部分 Rectangle 的成员变量，或者初始化全部的 Rectangle 的成员变量。如果没有提供初始值，构造器为任一成员变量提供一个默认值。例如：

```
public Rectangle() {
    this(0, 0, 0, 0);
}
```

这里的无参构造器调用带有 4 个参数的构造器，并提供 4 个 0 作为初始值；而两个参数的构造器调用 4 个参数的构造器，并提供两个 0 作为初始值。例如：

```
public Rectangle(int width, int height) {
    this(0, 0, width, height);
}
```

和前面几节的描述一样，编译器基于参数的数量和类型决定调用哪一个构造器。正如代码中所显示的，对另一个构造器的调用，必须是当前构造器的第一行代码。



6.7　控制对类的成员的访问

访问级别修饰符用来决定其他类是否能访问一个特定的字段或调用一个特定的方法。有如下两种级别的访问控制。

- 最高级别：public，或包级私有(package-private，没有指定修饰符)，用于修饰类。
- 成员级别：public、private、protected，或者包级私有(package-private，没有指定修饰符)，用于修饰类中的成员。

一个类可能会使用 public 修饰符声明，在这种情况下，类对在任何地方的其他类都是可见的。如果一个类没有修饰符（这是访问级别是默认的，即包级私有），那么这个类只能在它自己的包中可见。

在成员级别，也有 public 修饰符或无修饰符（包级私有），它们的含义与最高级别相同。对于成员，还有两个额外的访问修饰符 private 和 protected。private 修饰符说明成员只能在它所在的类的内部被访问。protected 修饰符说明成员只能在它所在的包中被访问到(包级私有)，另外，也可以被位于其他包中的该类的子类访问到。在成员级别，访问优先级是这样的：public > protected > package-private > private。如表 6.1 所示列出了每个访问控制修饰符所允许访问的成员。

表 6.1　访问级别

修 饰 符	类	包	子　类	全　部
public	是	是	是	是
protected	是	是	是	否
无修饰符	是	是	否	否
private	是	否	否	否

表 6.1 表明，一个类总是可以访问它自己的成员。访问级别从两个方面影响程序员：

- 当使用来自于另外一个途径的类时，如 Java 平台所提供的类，访问级别决定程序

员自己编写的类可以使用这些类的哪些方法。

- 当程序员自己编写了一个类时，可以决定类中的每一个成员变量和每一个成员方法的访问级别。

那么如何选择使用访问级别呢？如果 B 程序员要使用 A 程序员的类，A 程序员必须保证不要发生错误地使用这种情况。使用访问级别有助于这种控制：

- 对特定的成员，使用最严格的访问级别。除非有更好的理由，否则尽可能使用 private。
- 避免使用 public 字段，除非是定义常量。使用 public 会使得代码与特定的实现相联系，从而限制了代码改变的灵活性。



6.8 理解实例成员和类成员

在类中定义的字段和方法，不加特别声明的话，在创建实例对象时，在每一个实例对象中都会有这些字段的拷贝。但有时候，需要在同一个类的各个实例对象中共享同样的字段或方法，这就需要使用类成员。在本节，就将学习使用 static 关键字来创建属于类的字段和方法，而不是类的实例的字段和方法。

6.8.1 类变量

当从同一个类模板创建许多对象时，每一个对象都有它们自己不同的“实例变量”的副本。在汽车类 AutoCar 中，实例变量是 brand、exhaust 和 speed。每一个 AutoCar 对象中这些变量都有它自己的值，存储在不同的内存空间。

有时，需要对所有对象都通用的变量。这就需要使用 static 修饰符来实现。在声明中含有 static 修饰符的字段称为“静态属性”或“类变量”。它们与类相关联，而不是与对象相关联。该类的每一个实例对象都共享同一个类变量，这个类变量存储在内存中固定的位置。每一个对象都可以改变这个类变量的值，但是类变量不需要创建一个类的实例对象就可以对其进行操纵。

【例 6.8】类变量应用示例。

```
class  A {
     public static int a=0;      //声明一个静态变量 a（类变量 a）
     A(){
          a++;                   //每个对象生成时，都会调用这个构造器，对 a 自增
     }
}
public class  StaticVariableDemo {
     public static void main(String[] args){
          System.out.println("创建 A 的第一个对象前,a="+A.a);
          A a1 = new A();        //创建一个实例对象
          System.out.println("创建 A 的第二个对象前,a="+A.a);
          A a2 = new A();        //创建另一个实例对象
          System.out.println("创建 A 的第二个对象后,a="+A.a);
     }
}
```

打开记事本，输入以上代码，并保存为 Test.java 程序。编译并运行该程序，输出结果



如下。

```
创建 A 的第一个对象前,a=0
创建 A 的第二个对象前,a=1
创建 A 的第二个对象后,a=2
```

对照代码和输出结果，可以看出，每一次创建 A 类一个新的对象，都会调用 A 的构造器，在构造器中对类变量 a 自增 1。因为所有的对象共享同一个类变量，所以它们对类变量 a 的改变也累加。

考虑前面讲的 AutoCar 类。假设想要创建多个 AutoCar 类的对象并为每一个都赋予一个序列号，第一个对象从 1 开始。对每一个对象来说，ID 值是唯一的，因而它是一个实例变量。同时，还需要一个字段来记录总共有多少个 AutoCar 对象已经被创建了，这样就知道应该为下一个对象赋予什么样的 ID。这样的字段是与单个的对象不相干的，而是整个类作为一个整体。在这里，为 AutoCar 类添加一个类变量 numberOfAutoCars，其代码如下所示。

```
public class AutoCar{
     private String brand;                    //表示汽车的品牌
     private float exhaust;                   //表示汽车的排气量
     private int speed;                  //表示汽车的速度
     //添加一个实例变量，作为对象的 ID
     private int id;
     //添加一个类变量，用来统计实例化的 AutoCar 的对象的数量
     private static int numberOfAutoCars = 0;
     …
}
```

对于类变量的引用，可以直接通过类的名称，而无须再生成对象通过对象引用。例如：

```
AutoCar.numberOfAutoCars
A.a
```

当然也可以通过创建一个对象，通过对象引用。例如：

```
new AutoCar( ).numberOfAutoCars
```

但是直接通过类名加圆点运算符来引用，可以更清楚地看出这是个类变量。所以推荐对于类变量，尽可能地直接通过类名来引用。在 AutoCar 类中，可以使用构造器来设置 id 实例变量，并增加 numberOfAutoCars 类变量的值。代码如下所示。

```
public class AutoCar{
     private String brand;                    //表示汽车的品牌
     private float exhaust;                   //表示汽车的排气量
     private int speed;                       //表示汽车的速度
     private int id;                          //添加一个实例变量，作为对象的 ID
     private static int numberOfAutoCars = 0;
                        //添加一个类变量，用来统计实例化的 AutoCar 的对象的数量
     public AutoCar(String brand,float exhaust,int speed){     //构造器
          this.brand = brand;                 //在构造器中，初始化成员变量
          this.exhaust = exhaust;
          this.speed = speed;

          //增加 AutoCar 对象的数量并赋值给 ID
          id = ++numberOfAutoCars;
     }
     //新的方法，用来返回 ID 实例变量的值
```

```
    public int getID(){
        return id;
    }
    …
}
```

6.8.2 类方法

Java 语言支持类方法（静态方法），就像类变量（静态变量）一样。类方法在它们的方法声明中使用 static 修饰符，并且被调用时应使用类名，而不需要创建一个类的实例对象再调用。例如：

```
ClassName.methodName(args)
```

静态方法的一个普遍的作用，是用来访问静态字段。例如，在上面的 AutoCar 类中，再添加一个静态方法，用来访问 numberOfAutoCars 静态字段。

```
public static int getNumberOfAutoCars( ){
    return numberOfAutoCars;
}
```

并不是说所有的实例变量和方法与类变量和方法的结合都可以：

- 实例方法可以直接访问实例变量和实例方法。
- 实例方法可以直接访问类变量和类方法。
- 类方法可以直接访问类变量和类方法。
- 类方法不能直接访问实例变量或实例方法，要访问它们必须使用对象引用。还有，类方法不能使用 this 关键字，因为没有 this 的实例可供使用。

6.8.3 常量

修饰符 static 与修饰符 final 联合使用，还经常用于定义常量。final 修饰符说明这个字段的值不能改变。例如，下面的变量声明定义了一个常量 PI，它的值近似于圆周率π，在程序运行期间其值不能被改变。

```
static final double PI = 3.141592653589793;
```

使用这种方式所定义的常量不能被重新赋值，否则将会出现编译时错误。按照惯例，常量名称用大写字母。如果常量的名称由多个单词组成，单词之间用下画线“_”分隔。

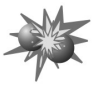
如果定义一个原始数据类型或一个字符串类型的常量，并且它们的值在编译时就知道，编译器会将代码中所有的常量用它们的值代替。这称为“编译时常量”。如果常量的值在外部世界被改变了（例如，通过立法规定π的值应该是 3.975），就需要重新编译任何使用了这个常量的类以获得当前新的值。

6.8.4 实例：AutoCar 类

在做了以上各节的修正以后，AutoCar 类现在应该是这样，它既有实例变量和实例方法，也有成员变量和成员方法。如例 6.9 所示。这样的类 AutoCar 就是一个完整的类的实现了。

【例 6.9】修改后的 AutoCar 类。

```
public class AutoCar2{
    private String brand;                    //表示汽车的品牌
```



```
    private float exhaust;              //表示汽车的排气量
    private int speed;                  //表示汽车的速度
    private int id;                     //添加一个实例变量，作为对象的 ID
    private static int numberOfAutoCars = 0;
                        //添加一个类变量，用来统计实例化的 AutoCar 的对象的数量
    public AutoCar2(String brand,float exhaust,int speed){
        this.brand = brand;                 //使用构造器参数赋初值
        this.exhaust = exhaust;
        this.speed = speed;
        //增加 AutoCar 对象的数量并赋值给 ID
        id = ++numberOfAutoCars;
    }
    public int getID( ){                    //获得 id 属性的方法
        return id;                          //返回 id 属性的值
    }
    public static int getNumberOfAutoCars( ){
                        //获得 AutoCar 的对象实例数量的方法
        return numberOfAutoCars;
    }
    public String getBrand(){               //获得属性 brand 的值
        return brand;
    }
    public float getExhaust(){              //获得属性 exhaust 的值
        return exhaust;
    }
    public int getSpeed(){                  //获得属性 speed 的值
        return speed;
    }
    public void setSpeed(int newSpeed){     //设置属性 speed 的值
        speed = newSpeed;
    }
}
```

在这个修改后的 AutoCar 类中，既声明有实例变量 brand 、exhaust、speed 和 id，也声明有静态的类变量 numberOfAutoCars。在构造器中完成对实例变量的初始化，并且每一次调用构造器创建一个新的实例对象时，都会将类变量 numberOfAutoCars 的值自增 1，以记录创建的对象数量。注意在这个类的定义中，实例方法可以访问类变量，但是类方法不能访问实例变量，否则就会出现编译时错误。



6.9　初始化字段

初始化字段，指的是在创建实例对象时，为其字段赋予初始值。其中实例变量可以在字段声明时进行初始化，也可以在构造器中进行初始化。在构造器中进行实例变量的初始化时，可以应用错误处理或其他逻辑。而要初始化类变量，Java 语言提供了“静态初始化块”。需要注意的是，在类的定义的开始声明字段并不是必需的，虽然这是最普遍的做法。只要是在它们被使用之前进行声明和初始化，都是可以的。

6.9.1　静态初始化块

一个“静态初始化块”是一个用花括号括起来的正常的代码块，以 static 关键字为前

导，用来对类中的静态变量进行初始化。其语法形式如下所示。

```
static{
     //需要进行初始化的代码
}
```

一个类中可以有许多静态初始化块，并且它们可以出现在类体的任何地方。运行时系统会保证静态初始化块会按照它们在源代码中出现的顺序被调用。可以通过私有静态方法来代替静态块，例如：

```
class Whatever {
    public static varType myVar = initializeClassVariable();
    private static varType initializeClassVariable() {
        //初始化代码
    }
}
```

其中，varType 指的是返回类型。在这里，声明一个私有静态（private static）方法 initializeClassVariable 来对类变量进行初始化，而不是使用静态初始化块。使用私有静态方法的好处是，当需要重复初始化类变量时，可以重用它。

6.9.2 初始化实例成员

如前面的示例所示，当有初始化值，并且初始化可以放在一行上时，经常在字段的声明中提供一个初始值，以初始化实例成员，如下面代码所示。

```
public class BedAndBreakfast{
     public static int capacity = 10;              //初始化为 10
     private boolean full = false;                 //初始化为 false
}
```

这样做是可以的。但是，这种形式的初始化因其过于简单而有一定的局限性。如果初始化要求一些逻辑（例如，错误处理或用一个 for 循环填充一个复杂的数组），这种简单的赋值就是不合适的。

通常，将初始化实例变量的代码放在构造器中。不过，有两个可供选择的方式来代替使用构造器初始化实例变量：初始化块和 final 方法。用于实例变量的初始化块看上去与静态初始化块相似，但是没有 static 关键字。

```
{
    //需要进行初始化的代码
}
```

Java 编译器复制初始化块到每一个构造器。因此，这种途径可以被多个构造器用来共享同一个代码块。另一种方式是使用 final 方法，因为一个声明为 final 的方法不允许在子类中被重写（这在接口和继承中解释）。下面是一个使用 final 方法对一个实例变量进行初始化的示例。

```
class Whatever {
    private varType myVar = initializeInstanceVariable();
    protected final varType initializeInstanceVariable() {
        //初始化代码
    }
}
```



其中 varType 是返回类型。如果子类想重用此初始化方法时，这特别有用。这个方法必须是 final()方法（最终方法），因为在实例初始化期间调用非 final()方法将会引起错误，所以在实例初始化期间对实例变量的初始化，必须调用 final()方法。

6.10　枚举类型

枚举类型是指其字段由一系列固定的常量组成的数据类型。在生活中，常见这种类型的数据，例如，表示方向的值，只能是东、西、南、北，表示一周中的天数，只能是星期一、星期二、星期三、星期四、星期五、星期六、星期日。因为是常量，所以枚举类型的字段要用大写字母表示。

在 Java 语言中，使用关键字 enum 定义一个枚举类型。例如，可以使用枚举类型来指定一周中的天数，如下所示。

```
public enum Day {
     星期日, 星期一, 星期二, 星期三, 星期四, 星期五, 星期六
}
```

在任何时候，如果需要代表一系列固定的常量，就应该使用枚举类型。在两种情况下，尽可能使用枚举类型，一种是自然的枚举类型，例如，表示太阳系中的行星或者员工的性别，这是一组固定的值，所以在程序中应该使用枚举类型来表示。另一种是程序在编译的时候，已经知道某个数据所有可能的值，例如菜单、命令行标记等，也要尽可能使用枚举类型。

例如，在扑克牌游戏中，花色也是固定的，只有 4 种颜色：方块、梅花、红心和黑桃。所以在一个扑克游戏的程序中，如果要定义表示扑克牌花色的类型，就可以使用枚举，如下所示。

```
public enum Suit {
     DIAMONDS,                          //方块
     CLUBS,                             //梅花
     HEARTS,                            //红心
     SPADES                             //黑桃
}
```

【例 6.10】枚举示例程序。演示如何使用上面所定义的名为 Day 的枚举类型。

```
public class EnumDemo{
     Day day;                                   //声明 Day 枚举类型的变量 day
     public EnumDemo(Day day) {                 //构造器，传递进来一个 Day 类型的参数
          this.day = day;
     }
     public void tellFeeling() {                //匹配每一天，并输出感受
          switch (day) {
               case 星期一: System.out.println("星期一都不令人喜欢.");
                         break;
               case 星期五: System.out.println("星期五比较好，有盼头.");
                         break;
               case 星期六:
               case 星期日: System.out.println("周末真令人愉快.");
                         break;
               default:  System.out.println("一周的中间一般般，不好也不坏.");
                             break;
          }
```

```
    }
    public static void main(String[] args) {
        EnumDemo firstDay = new EnumDemo (Day.星期一); //创建一个新的对象实例
        firstDay. tellFeeling ();     //调用其 tellFeeling 方法
        EnumDemo thirdDay = new EnumDemo (Day.星期二); //创建另一个新的对象实例
        thirdDay. tellFeeling ();     //同样调用其 tellFeeling 方法
        EnumDemo fifthDay = new EnumDemo (Day.星期五);//创建第 3 个对象实例
        fifthDay. tellFeeling ();
        EnumDemo sixthDay = new EnumDemo (Day.星期六);//创建第 4 个对象实例
        sixthDay. tellFeeling ();
        EnumDemo seventhDay = new EnumDemo (Day.星期日);
                                                    //创建第 5 个对象实例
        seventhDay. tellFeeling ();
    }
}
```

使用如下命令依次编译 Day.java 和 EnumDemo.java。

```
javac Day.java
javac EnumDemo.java
```

程序运行结果如下。

```
星期一都不令人喜欢.
一周的中间一般般，不好也不坏.
星期五比较好，有盼头.
周末真令人愉快.
周末真令人愉快.
```

Java 语言中的枚举类型要比其他语言中的枚举类型要强大得多。在 Java 中，enum 声明定义了一个类（称为“枚举类型”）。枚举类的类体中可能包括方法和其他字段。当编译器创建一个枚举时，它会自动地添加一些专门的方法。例如，它会添加一个静态的 values() 方法，该方法会按声明的顺序返回一个包含枚举中所有值的数组。

values()方法通常与 for-each 结构一起使用，以迭代一个枚举类型的值。例如，要迭代一星期中的每一天，使用如下代码。

```
for(Day day : Day.values()){
    System.out.println(“今天是” + day);
}
```

所有的枚举都隐含地继承自 java.lang.Enum。因为 Java 不支持多重继承，所以一个枚举类型不能再从任何其他类继承。

在下面的示例中，定义了一个 Planet 枚举类型，代表太阳系中的行星。Planet 中使用了代表质量和半径的常量属性。

【例 6.11】枚举示例程序。在程序中，输入你在地球上的体重，并计算和输出在各个行星上时的体重。

```
public enum Planet {
    水星(3.303e+23, 2.4397e6),                  //水星
    金星(4.869e+24, 6.0518e6),                  //金星
    地球(5.976e+24, 6.37814e6),                 //地球
    火星(6.421e+23, 3.3972e6),                  //火星
    木星(1.9e+27,   7.1492e7),                  //木星
    土星(5.688e+26, 6.0268e7),                  //土星
```



```
    天王星(8.686e+25, 2.5559e7),                    //天王星
    海王星(1.024e+26, 2.4746e7);                    //海王星,注意这里要以分号结尾
    private final double mass;                      //代表质量，单位为千克（公斤）
    private final double radius;                    //代表半径，单位为米
    Planet(double mass, double radius) {            //构造器
        this.mass = mass;
        this.radius = radius;
    }
    private double mass()   { return mass; }     //获得质量
    private double radius() { return radius; } //获得半径
    // 通用的万有引力常量
    public static final double G = 6.67300E-11;
    double surfaceGravity() {                       //获得地心引力（重力）
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {    //获得地表重量
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {
        if(args.length<1){                          //如果没有命令行参数，则返回
             System.out.println("请在命令行中输入参数!");
             return;
         }
double earthWeight = Double.parseDouble(args[0]);
                                  //通过命令行参数传递地球上的体重
        double mass = earthWeight/地球.surfaceGravity(); //获取质量大小
        for (Planet p : Planet.values())     //迭代输出在每个行星上的体重
           System.out.printf("你在%s上重%f公斤。%n",p,p.surfaceWeight(mass));
    }
}
```

使用如下命令编译并运行 Planet 程序，注意在运行时要在命令行传递进一个表示在地球上的体重的参数。

```
javac Planet.java
java Planet 78
```

程序输出结果如下。

```
你在水星上重 29.465094 公斤。
你在金星上重 70.589930 公斤。
你在地球上重 78.000000 公斤。
你在火星上重 29.541500 公斤。
你在木星上重 197.383487 公斤。
你在土星上重 83.149212 公斤。
你在天王星上重 70.599922 公斤。
你在海王星上重 88.789590 公斤。
```

正如上面的示例代码所示，每一个枚举常量使用表示质量和半径的参数值来声明。当常量被创建时，这些值被传递给构造器。Java 语言要求常量要首先定义，优先于任何其他的字段和方法。另外，如果有其他的字段和方法，枚举常量的列表必须以分号“;”结束。

一个枚举类型的构造器必须为包级私有的或私有的（private）。它自动地创建在枚举体一开始定义的常量。不能人为地调用枚举的构造器。除了属性和构造器外，Planet 还定义了一些方法，用来计算每个行星的地心引力和在行星上对象的重量。

6.11 标注

标注（Annotations）就是代码里的标记，它提供与程序有关的数据，但是标注本身不是程序一部分。标注不对其所注解的代码的操作有直接的影响。在有的书中，将标注也称为注解。通过使用标注，程序开发人员可以在不改变原有逻辑的情况下，在源文件中嵌入一些补充的信息。代码分析工具、开发工具和部署工具可以通过这些补充信息进行验证或进行部署。

6.11.1 标注的用法

简单地说，标注就是代码里的标记，这些标记在类加载、运行时或编译的时候可以被解释，但是不对程序的运行产生直接的影响。标注有很多用法，包括：

- 为编译器提供信息。编译器可以使用标注来检测错误或禁止警告。
- 在编译和部署时处理。软件开发工具可以处理标注信息以生成代码、XML 文件等。
- 在运行时处理。有些标注在运行时可以被检查并使用。

从某些方面看，标注就像修饰符一样被使用，并应用于包、类型、构造方法、方法、成员变量、参数、本地变量的声明中。标注可以应用到程序的类的声明、字段的声明、方法的声明及其他程序元素的声明中。按惯例标注要出现在它自己所在行的第一行，并且可以包括带有名称或未命名值的元素，如下所示。

```
@Author(
   name = "辛立伟",
   date = "2008-12-12"
)
class MyClass() { }
```

或者

```
@SuppressWarnings(value = "unchecked")
void myMethod() { }
```

如果只有一个为值的元素，可以省略名称，如下所示。

```
@SuppressWarnings("unchecked")
void myMethod() { }
```

此外，标注也可以没有元素，这时可以省略圆括号，如下所示。

```
@Override
void myMethod() { }
```

6.11.2 文档标注

许多标注用来代替应该出现在代码中的注释。假设一个软件以传统的方式开始每个类，通常使用大量的注释来提供重要的信息，如下所示。

```
public class B extends A{
    //作者：辛立伟
    //日期：2008-12-12
    //当前版本：V1.0
    //最后修改日期：2008-12-28
    //参与者：张三，李四，赵小六
```



```
    //类体代码…
}
```

如果要使用标注来添加同样的元数据信息，必须首先定义“标注类型”。定义标注类型的语法如下所示。

```
@interface AuthorInfo {
    String 作者();
    String 日期();
    int 当前版本() default 1;
    String 最后修改日期() default "N/A";
    String[ ] 参与者();                    //注意，这里使用数组
}
```

标注类型的定义有点类似于接口的定义，使用一个’@’字符作为关键字 interface 的前导字符。’@’（发“at”音）代表标注类型。实际上，标注类型是接口的一种。

上面定义的标注的标注体内容包括“标注类型元素”声明，看上去与方法的声明类似。注意它们可以定义可选的默认值。一旦定义了标注类型，就可以使用该类型的标注了，像下面这样填充值进去。

```
@ AuthorInfo (
    作者 = "辛立伟",
    日期 = "2008-12-12",
    当前版本 = 6,
    最后修改日期 = "2008-12-28",
    参与者 = {"张三", "李四", "赵小六"}    //注意使用数组符号
)
public class B extends A {
    //类体的代码…
}
```

如果要使在“@ AuthorInfo”中的信息出现在 Javadoc 生成的文档中，必须使用“@Documented”标注注释“@AuthorInfo”的定义，如下所示。

```
import java.lang.annotation.*;          //导入这个包才能使用@Documented
@Documented
@interface AuthorInfo {
    String 作者();
    String 日期();
    int 当前版本() default 1;
    String 最后修改日期() default "N/A";
    String[ ] 参与者();                    //注意，这里使用数组
}
```

6.11.3　预定义标注

Java 语言规范本身在 java.lang 包中预定义有三种标准的标注类型，分别是：@Deprecated、@SuppressWarnings 和@Override。下面逐一介绍这三种标准标注类型。

1．@Deprecated 不提倡标注

当一个类型或者类型成员使用@Deprecated 修饰的话，编译器将不提倡使用这个被标注的程序元素。而且这种修饰具有一定的“延续性”：如果在代码中通过继承或者覆盖的

方式使用了这个过时的类型或者成员，虽然继承或者覆盖后的类型或者成员并不是被声明为“@Deprecated”，但编译器仍然要报警。

当一个程序使用带有“@Deprecated”标注的方法、类或字段时，编译器会生成一个警告信息。当一个元素是不提倡使用的时，就应该使用 Javadoc 的“@deprecated”标签将其收入文档中，如下面的代码所示。

```
//Javadoc 注释
    /**
     * @deprecated
     * 解释为什么反对它
     */
    @Deprecated
    static void deprecatedMethod() { }          //被反对的方法声明
```

在 Javadoc 注释和在标注中使用“@”符号是不一致的，它们仅是概念上的相关。另外要注意，Javadoc 标签以一个小写字母“d”开头，而标注以一个大写的“D”开头。例如下面的代码：

```
public interface House {
    /**
     * @deprecated
     * 不提倡使用 open 方法，首选使用 openFrontDoor 或 openBackDoor 方法.
     */
    @Deprecated
    public void open();
    public void openFrontDoor();
    public void openBackDoor();
}
```

在上面的接口中，提倡使用更明确的 openFrontDoo()r 或 openBackDoor()方法，而不提倡使用 open()方法。如果在程序中使用了 open()方法，那么根据标注，编译器就会生成一个警告信息。例如，再定义一个实现了上述 House 接口的类 MyHouse，并保存为 MyHouse.java 文件。定义如下。

```
public class MyHouse implements House {
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

这时使用如下命令编译 MyHouse 类

```
javac MyHouse.java
```

会出现如下警告信息：

```
注意：MyHouse.java 使用或覆盖了已过时的 API。
注意：要了解详细信息，请使用 -Xlint:deprecation 重新编译。
```

如果不想编译时出现这样的警告提示信息，可以使用如下方法：

```
public class MyHouse2 implements House {
    //从 house 接口中继承文档注释标签
    @Deprecated
    public void open() {}
    public void openFrontDoor() {}
```



```
    public void openBackDoor() {}
}
```

也就是说，用“@Deprecated”标注 open()方法的实现，那么在编译 MyHouse2 类时，就不会出现警告提示信息了。

2．@Override 重载标注

使用“@Override”标注，用来“告诉”编译器，下面的元素要重载在父类中声明的一个元素。它说明了被标注的方法重载了父类的方法，起到了断言的作用。例如：

```
// 标注要被覆盖的超类的方法
@Override
int overriddenMethod() { }
```

虽然重载一个方法的时候并不要求使用这个标注，但是使用它有助于防止错误。如果使用“@Override”标注的方法不能正确地覆盖其超类中的方法，那么编译器就会生成一个错误。例如下面的代码所示。

```
interface Closable {
    void close();
}

class File implements Closable {
    @Override
    public void close() {
        //... 关闭文件...
    }
}
```

在编译类 File 时，编译器会生成一个错误，提示 File.close()方法并没有覆盖任何其超类的方法。因为它并没有覆盖 Closable.close 方法，而是实现了接口中的 close()方法。“@Override”标注常常在试图覆盖父类方法而又写错了方法名时发挥作用。

3．@SuppressWarnings 禁止警告标注

“@SuppressWarnings”用于有选择地关闭编译器对类、方法、成员变量、变量初始化的警告。在 Java 5.0 中，Sun 提供的 javac 编译器提供了-Xlint 选项来使编译器对合法的程序代码提出警告，此警告从某种程度上代表了程序错误。例如，当使用一个泛型集合类（generic collection 类）而又没有提供其类型时，编译器将提示出"unchecked warning"的警告。

通常当这种情况发生时，就需要查找引起警告的代码。如果它真的表示错误，就需要纠正它。例如如果警告信息表明代码中的 switch 语句没有覆盖所有可能的 case，那么就应增加一个默认的 case 来避免这种警告。

类似地，有时无法避免这种警告，例如，使用必须和非泛型的旧代码交互的泛型集合类时，不能避免这个“unchecked warning”。此时“@SuppressWarning”就要派上用场了。在调用的方法前增加“@SuppressWarnings”修饰，告诉编译器停止对此方法的警告。

SuppressWarning 有一个类型为 String[]的成员，这个成员的值为被禁止的警告名。对于 javac 编译器来讲，被-Xlint 选项有效的警告名也同样对“@SuppressWarings”有效，同时编译器会忽略掉无法识别的警告名。因此，当不想出现某些警告时，使用“@Suppress Warnings”标注用来“告诉”编译器，阻止生成某些指定的警告。如下面的代码所示。

```
//使用了一个不提倡使用的方法，并告诉编译器不要生成警告
@SuppressWarnings("deprecation")
```

```
void useDeprecatedMethod() {
    objectOne.deprecatedMethod();   //不提倡使用的信息被阻止
}
```

如果使用了一个不提倡使用的方法，正常情况下编译器应该生成一个警告。但在上面的代码示例中，标注所引起的警告被阻止了。

例如，在前面声明的 House 接口中，使用“@Deprecated”标注了不提倡使用的 open()方法，那么在编译 House 接口的实现类 MyHouse 时，就会出现警告提示信息。如果要阻止编译时出现这样的警告提示信息，除了可以同样使用“@Deprecated”标注 open()方法的实现之外，也可以使用“@SuppressWarnings("deprecation")”标注禁止出现“不提倡”的警告提示信息，如以下代码所示。

```
public class MyHouse3 implements House {
    @SuppressWarnings("deprecation")
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

每一个编译器的警告信息都属于一种类型。在 Java 语言规范中列出了两类：不提倡的和未检查的。上面使用的是不提倡的编译器警告。而当连接泛型出现以前编写的代码时，会出现“未检查的”警告。要阻止超过一种类型的警告，使用如下的语法：

```
@SuppressWarnings({"unchecked","deprecation"})
```

6.11.4 标注处理

对于标注更高级的使用是自己编写一个“标注处理器”。标注处理器可以处理 Java 程序并根据其标注采取相应的动作。例如，生成辅助代码、帮助程序员按照预定义的模式创建模板代码。在 JDK 中包括有一个标注处理工具，名为“apt”。在 JDK6.0 中，apt 的功能已经是 Java 编译器的标准组成部分了。

要使标注信息在运行时可用，其本身必须有一个@Retention(RetentionPolicy.RUNTIME)标注，如下所示。

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationForRuntime {
   //为运行时处理提供信息的元素
...
}
```

6.12 小结

本章主要学习了类和对象的相关概念，包括类的声明、类的实例（即对象）。本章所讲的知识是 Java 面向对象语言中最基本的概念，所涉及的类和声明、对象的构造、访问控制修饰符都是读者学习的重点。其中的构造器的工作过程、访问控制和 this 关键字的使用是本章学习的难点。

通过本章的学习，读者应该对类和对象有深刻的理解，并能正确而熟练地使用类和对象。在第 7 章中，将向读者介绍面向对象另一个非常重要的概念：继承。

第 7 章 继承

继承是面向对象语言的三大特征之一。在本章中，将学习怎样从一个类派生出另一个类，也就是说，一个子类如何从其父类继承字段和方法。还将学习到所有的类都派生自 Object 类，以及如何修改从父类继承过来的子类的方法。



7.1 继承概述

在面向对象的程序设计中，继承是不可或缺的一部分。继承是面向对象程序设计语言的基本特征之一。通过继承，可以快速地创建新的类，可以实现代码的重用，提高程序的可维护性，节省大量创建新类的时间，提高开发效率和开发质量。

7.1.1 继承的概念

在第 6 章中讲如何定义类时，已经提到过继承。在 Java 语言中，一个类可以从其他的类派生出来，从而继承其他类的字段和方法。从另外的类派生过来的类被称为子类（或者派生类、扩展类、子孙类）。用来派生子类的类被称为超类（或基类、父类）。

在 Java 中，除了 Object 类没有超类，其他每个类都有一个并且只有一个直接的超类（这称为 Java 的“单继承”）。在没有任何其他显式的超类的情况下，每一个类都隐含是 Object 类的子类。

类可以从继承自其他类的类派生过来，也就是说，用来派生子类的类也可以是其他类的子类，并依此类推，最终派生自最顶层的类 Object。这种情况下，可以说 Java 中的所有类都是 Object 类的后代类。

继承的思想很简单，但功能却很强大：当程序员想要创建一个新的类时，如果已经存在一个含有他所想要的代码的类，那么他可以从已经存在的类派生出新的类，快速、简单、安全。这时候，就可以重用已经存在的类的字段和方法而不必编写（并测试）它们。

一个子类从它的超类继承所有的成员，包括字段、方法和内嵌类。需要注意的是，构造器不是成员，所以构造器不会被子类继承，但是可以从子类中调用超类的构造器。

7.1.2 Java 平台中的类层次

在 java.lang.package 包中定义的 Object 类，定义和实现了对所有的类来说最通用的行

为。在 Java 平台中，许多类都直接继承自 Object，其他类再从这些类继承，依此类推，形成类的一个层次，如图 7.1 所示。

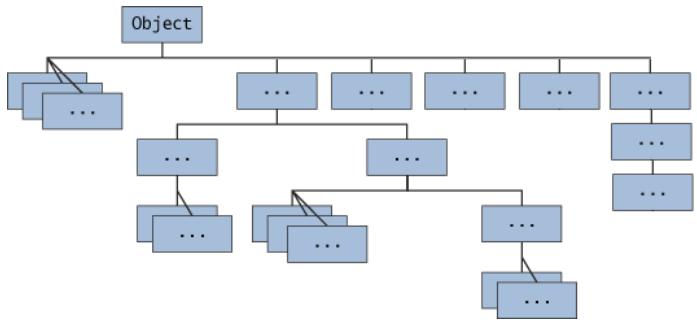


图 7.1　Java 平台中所有的类都继承自 Object

在类层次的顶点是 Object 类，Object 类提供了对所有类的高度概括。在类层次底部的类则提供更加特定的行为。

7.1.3　一个继承的示例

在第 6 章中，提到一个 AutoCar 类的实现。AutoCar 类是对各种汽车的一个基本概括，含有表示其品牌、排气量和速度的字段及对这些字段属性进行读取的相应方法。其代码实现如下所示。

```
public class AutoCar{
    //定义三个私有字段，代表 AutoCar 对象的三个属性
    private String brand;                                    //表示汽车的品牌
    private float exhaust;                                   //表示汽车的排气量
    private int speed;                                       //表示汽车的速度
    //定义一个带参的构造器，用来初始化字段
    public AutoCar(String brand,float exhaust,int speed){
        this.brand = brand;
        this.exhaust = exhaust;
        this.speed = speed;
    }
    //获得属性 brand 的值
    public String getBrand(){
        return brand;
    }
    //获得属性 exhaust 的值
    public float getExhaust(){
        return exhaust;
    }
    //获得属性 speed 的值
    public int getSpeed(){
        return speed;
    }
    //设置属性 speed 的值
    public void setSpeed(int newSpeed){
        speed = newSpeed;
    }
}
```



这个 AutoCar 类是泛指，代表广义概念上的汽车。现在，想要创建一个代表更具体的卡车的 Car 类，最简单并且最合适的途径，是从 AutoCar 类派生出来，然后在子类中只需要增加新的特性（字段或方法）即可。

【例 7.1】由 AutoCar 类派生出来更加具体的 Car 子类。

```
public class Car extends AutoCar{

    //Car 子类中增加一个字段，代表载重量
    public int weight;

    //Car 子类有一个构造器
    public Car(int startWeight,String originalBrand,float originalExhaust,
int startSpeed){
        super(originalBrand,originalExhaust,startSpeed); //调用超类的构造器
        weight = startWeight;                    //初始化子类中新增加的属性
    }

    //Car 类增加一个方法
    public void setWeight(int newWeight){
        weight = newWeight;
    }
}
```

卡车类 Car 继承了 AutoCar 类的所有字段和方法，并增加了一个代表载重的字段 weight 和一个设置该字段的方法 setWeight()。除了构造器之外，就好像重新写一个全新的 Car 类一样，Car 类带有 4 个字段和 5 个方法，包括了从父类 AutoCar 继承过来的字段和方法。实际上，在创建新的 Car 类时，并不需要全部重写这些字段和方法。

当超类 AutoCar 中的方法很复杂，并且需要耗费大量时间进行调试时，使用继承特别有用，可以重用已经经过测试的安全健壮的代码，而不需要重新编写，重新测试。

7.1.4　在子类中可以做的事情

不论子类位于哪个包中，它都继承其超类所有的 public（公共的）和 protected（受保护的）成员。如果子类和超类在同一个包中，它还继承超类的包级私有(package-private)的成员（包级私有成员，指的是类中不带修饰符的成员和方法，默认对其访问权限仅限于同一个包中）。在子类中，既可以不加修改地使用继承过来的成员，也可以替换、隐藏它们，或者使用新的成员对其进行补充。

- 继承过来的字段可以像任何其他字段一样被直接使用。
- 在子类中可以声明一个与超类中同名的新的字段，从而“隐藏”超类中的字段（不推荐使用）。
- 可以在子类中声明一个在超类中没有的新的字段。
- 从超类继承过来的方法可以直接使用。
- 可以在子类中编写一个与超类当中具有相同签名的新的实例方法，这称为“方法重写”或“方法覆盖”。
- 可以在子类中编写一个与超类当中具有相同签名的新的静态方法，从而“隐藏”超类中的方法。

- 可以在子类中声明一个超类当中没有的新的方法。
- 可以在子类中编写一个调用超类构造器的子类构造器，这既可以隐式地实现，也可以通过使用关键字 super 来实现。

在后面的各节中将会详述这些内容。

7.2 从子类中访问父类的私有成员

子类继承其父类的所有 public（公共的）和 protected（受保护的）成员，但不继承其父类中的 private（私有的）成员。例如，下面的示例中，类 A 含有一个公共变量 value，在其子类 B 中就可以直接访问这个字段。

【例 7.2】在子类中访问父类的 public 成员变量。

```
//父类 A
class A{
    public int value;                                    //声明一个公共变量
}
//A 的子类 B
class B extends A{
}
//主程序 Test
public class ExtendsDemo1{
    public static void main(String[ ] args){
        B b = new B( );                                  //创建子类 B 的一个实例对象
        System.out.println(“可以在子类 B 中直接访问它的父类 A 中的公共字段 value：”
+ b.value);
    }
}
```

编译并运行这个程序，输出结果如下。

```
可以在子类 B 中直接访问它的父类 A 中的公共字段 value：0
```

如果将上例中类 A 的字段 value 前的访问控制修饰符改为 private，再一次编译此程序，会出现编译错误信息，如下所示。

```
Test.java:17: value 可以在 A 中访问 private
        System.out.println("可以在子类 B 中直接访问它的父类 A 中的公共字段 value："
+ b.value);
                                                          ^
1 错误
```

由此可以看出，因为子类不继承其父类中的私有成员，所以在子类中不能直接访问其父类中的私有成员。如果想在子类中访问到父类中的私有字段，那么需要在超类中提供用来访问其私有字段的 public 或 protected 方法，然后子类就可以使用这些方法来访问相应的字段。

【例 7.3】修改上面的示例程序，将类 A 的字段 value 改为私有的，然后提供一个访问该字段的公共方法 getValue。其代码如下所示。

```
//父类 A
class A{
    private int value;                                   //声明一个私有变量
    //提供一个公共方法，对外界提供一个访问其私有字段的接口
    public int getValue(){
```

```
            return value;                              //返回属性值
        }
    }
    //A 的子类 B
    class B extends A{
    }
    //主程序 Test
    public class ExtendsDemo2{
        public static void main(String[ ] args){
            B b = new B( );                            //创建子类 B 的一个实例对象
            System.out.println("在子类 B 中通过其父类 A 提供的公共访问接口访问 A 中的私有字段 value:"
                                +b.getValue());
        }
    }
```

编译并运行这个程序，输出结果如下。

```
在子类 B 中通过其父类 A 提供的公共访问接口访问 A 中的私有字段 value:0
```

在上面这个程序中，子类 B 继承了其父类的 public 方法 getValue()，因此可以通过调用 getValue()方法间接地访问其父类的私有字段。而对类 A 来说，通过将其字段封装为私有的，并提供公共访问接口，很好地保护了其私有数据。

7.3　对象类型转换

在 Java 中，一个对象的类型可以看做是其自身所属类的类型，也可以看做是其超类的类型。那么在其自身所属类型和其超类类型之间，就会发生对象类型转换。对象类型转换表示在允许继承和实现的对象中，使用一个类型的对象来替代另一个类型。

7.3.1　隐式对象类型转换

说到一个对象的数据类型，指的是实例化该对象的类。例如，当一个小朋友用手指着一个美丽的天鹅告诉妈妈："这是个美丽的天鹅"，那么如果从面向对象的角度来理解，可以理解为，小朋友所指的具体的鸟，其类型是天鹅。如果有下面这行代码：

```
public Car myCar = new Car( );
```

那么，可以说 myCar 这个对象是属于 Car 类型的。在上面的代码行中，Car 是 AutoCar 和 Object 的子孙类，因此一个 Car 既是一个 AutoCar，还是一个 Object，它可以被用于任何 AutoCar 或 Object 对象被调用的地方。

例如，张晓明（对象）是个中学生（类），也可以说张晓明是个学生（中学生、小学生、大学生等类的父类），凡是学生或中学生能做的事情，张晓明都能做，如学生要考试，属于学生的中学生当然也要考试，属于学生和中学生的张晓明当然也要考试。

可以这样理解，在可以使用超类的地方，都可以使用子类替代，因为子类继承了其超类所有的成员。换句话说，在代码中，可以定义一个父类的引用变量，实际指向一个子类。例如，下面的代码是允许使用的。

```
Object car = new Car();
```

或者

```
AutoCar car = new Car();
```

这里，car 既是一个 Object(或 AutoCar)类型，又是一个 Car 类型，这称为“隐式类型转换”。即将 Car 类型隐含地转换为其父类类型，因为子类继承了父类中所有的成员，凡能调用父类的地方，都可以使用子类。

7.3.2 强制对象类型转换

但是反过来就不成立：一个 AutoCar 可能是一个 Car，但并不一定是一个 Car。例如，在上面的例子中，张晓明是中学生，是学生。但反过来，说到学生或中学生时，并不一定就是指张晓明。同样，一个对象可能是，但不一定就是 AutoCar 或 Car。例如，将上一节的代码反过来写，如下面代码所示。

```
AutoCar car = new Car( );
Car myCar = car;
```

那么，在编译的时候将会得到一个编译时错误，因为在第二行中，编译器无法判断出 car 是一个 Car 类型。但是，可以通过“强制类型转换”来“告诉”编译器，car 引用变量引用的是一个 Car 类型的：

```
Car myCar = (Car)car;
```

在这种强制类型转换中，编译时编译器会进行运行时检查，会发现 car 引用变量已经被赋予了一个 Car 类型的对象，因此编译器就会安全地判断出 car 是一个 Car 类型。如果在运行时 car 不是一个 Car 类型，那么就会抛出一个异常。

7.3.3 使用 instanceof 运算符

使用强制类型转换时，如果转换的类型不合适，编译时会抛出异常。但对于用户来讲，这种异常信息经常让人很困惑，所以最合适的做法是在对某个对象进行强制类型转换之前，先对其类型进行判断。判断对象的类型使用 instanceof 运算符。

使用 instanceof 运算符对一个特定对象的类型进行逻辑判断，可以防止因为不合适的类型转换而产生的运行时错误。例如，将上一节强制类型转换的代码改写如下代码：

```
if(car instanceof Car){
    Car myCar = (Car)car;
}
```

在上面的代码中，instanceof 运算符检查 car 引用变量指向一个 Car 实例对象，所以可以放心地进行类型转换，而不必担心会抛出运行时异常。

 7.4 覆盖和隐藏超类中的方法

子类继承了父类中的所有成员，包括父类中的方法。然而，在某些情况下，子类中该方法所表示的行为与其父类中该方法所表示的行为不完全相同。例如，在父类 AutoCars 中，定义了加速这个行为，而在其不同的子类中，加速的实现是不相同的：卡车有卡车的加速行为的实现，吉普车有吉普车加速行为的实现（如四轮驱动）。这时，需要在子类中覆盖或隐藏其父类中的该方法。



7.4.1　覆盖超类中的实例方法

当一个子类中的一个实例方法具有与其超类中的一个实例方法相同的签名（指名称、参数个数和类型）和返回值时，称子类中的方法“重写”或“覆盖”了超类的方法。

【例 7.4】派生类中覆盖超类中的实例方法示例。

```
// 父类 A
class A{
    //表示问候的公共方法
    public void sayHello( ){
        System.out.println("Hello,everyone! ");         //输出英文问候
    }
    //表示道别的公共方法
    public void sayBye( ){
        System.out.println("GoodBye,everyone! ");  //输出英文道别
    }
}
// A 的子类 B
class B extends A{
    //用中文问候
    public void sayHello( ){
        System.out.println("大家早上好! ");           //输出中文问候
    }
}
// 主程序 Test
public class ExtendsDemo3{
    public static void main(String[ ] args){
        B b = new B( );              //创建子类 B 的一个实例对象，使用默认构造器
        b.sayHello( );               //调用子类中重写的方法
        b.sayBye( );                 //调用父类中的方法
    }
}
```

编译并运行该程序，输出结果如下。

```
大家早上好!
GoodBye,everyone!
```

在上面的示例程序中，子类 B 中定义了一个与其父类 A 中具有相同签名的方法 sayHello()。当在子类 B 中调用 sayHello()方法时，调用的是在子类中定义的该方法，这称为子类的 sayHello()方法覆盖了父类中的 sayHello()方法。而道别的方法 sayBye()没有被覆盖，所以在子类 B 中调用的是从其父类 A 继承的该方法。

子类重写超类方法的这种能力，允许一个类继承自行为“足够相似”的超类，并根据需要去修改行为。重写的方法具有与其所覆盖的方法相同的名称、参数数量和类型、返回值。

7.4.2　协变覆盖

一个重写的方法还能够返回其所覆盖的方法返回类型的子类型。这称为“协变（covariant）返回类型”。这是 Java 5.0 中开始的新特性。协变返回模式比较整洁、安全并且易于维护，可以避免进行强制类型转换。

例如，以 Object 类的 clone 方法为例（在 7.8 节会讲到），Object 类是 Java 中所有类的

间接超类，它的 clone()方法返回自身对象的一个拷贝。

```
class Object {
    ...
    public Object clone() { ... }
}
```

在 Java 5.0 以前，如果子类需要重写 clone()方法，必须像下面这样写代码：

```
class Point {
    public int x;                          //声明实例成员 x
    public int y;                          //声明实例成员 y
    //构造器
    public Point(int x, int y) {
        this.x=x;                          //初始化实例成员
        this.y=y;
    }
    //覆盖 Object 类中的 clone 方法，返回 Point 对象的一个拷贝，但返回类型为 Object
    public Object clone() {
        return new Point(x,y);             //创建并返回一个 Point 对象的引用
    }
}
```

虽然在上面的 Point 类里，clone()方法总是返回一个 Point 类型的对象，但却必须把返回类型写成 Object（Point 类的隐含父类是 Object），在外部使用 clone()方法时也必须使用恼人的强制类型转换。

```
Point p1 = new Point(23,35);               //创建一个 Point 对象 p1
Point p2 = (Point)p1.clone();              //使用 clone 方法获得 p1 的一个拷贝
```

在 Java5.0 以后，就可以利用新的协变覆盖规则，像下面这样编写代码：

```
class Point {
    public int x;                          //声明实例成员 x
    public int y;                          //声明实例成员 y
    //构造器
    public Point(int x, int y) {
        this.x=x;                          //初始化实例成员
        this.y=y;
    }
    //覆盖 Object 类中的 clone 方法，返回 Point 对象的一个拷贝
    public Point clone() {
        return new Point(x,y);             //创建并返回一个 Point 对象的引用
    }
}
```

这样，就可以直接使用“Point p2 = p1.clone(); ”，而不用强制类型转换了。

```
Point p1 = new Point(23,35);               //创建 Point 实例对象并初始化
Point p2 = p1.clone();                     //调用其 clone 方法克隆一个新的对象
```

7.4.3 隐藏超类中的类方法

如果一个子类定义了一个类方法（静态方法），而这个类方法与其超类中的一个类方法具有相同的签名（指名称、参数个数和类型）和返回类型，那么说，在子类中的这个类



方法“隐藏”了超类中的该类方法。

在隐藏和覆盖之间的区别有很重要的意义。当调用被覆盖的方法时，调用的版本是子类中的方法。当调用被隐藏的方法时，调用的版本取决于是从超类中调用的还是从子类中调用的。下面看一个包含两个类的例子。

【例 7.5】超类中隐藏方法的调用示例。实现步骤如下。

① 编写第一个类是 Animal，它包含一个实例方法和一个类方法。

```
public class Animal {
    public static void testClassMethod() {              //静态的类方法
        System.out.println("Animal 中的类方法.");
    }
    public void testInstanceMethod() {                  //定义实例方法
        System.out.println("Animal 中的实例方法.");
    }
}
```

② 编写第二个类，是 Animal 类的子类，名为 Cat 类。

```
public class Cat extends Animal {
    public static void testClassMethod() {              //定义静态的类方法
        System.out.println("Cat 中的类方法.");
    }
    public void testInstanceMethod() {                  //定义实例方法
        System.out.println("Cat 中的实例方法.");
    }
}
```

③ 再编写一个用来测试的程序 ExtendsDemo4.java。

```
public class ExtendsDemo4{
    public static void main(String[] args) {
        Cat myCat = new Cat();                  //创建 Cat 类的一个实例对象
        Animal myAnimal = myCat;                //隐式对象类型转换
        Animal.testClassMethod();               //调用 Animal 类的类方法
        myAnimal.testClassMethod();             //调用 myAnimal 对象的类方法
        Cat.testClassMethod();                  //调用 Cat 类的类方法
        myAnimal.testInstanceMethod();          //调用 myAnimal 对象的实例方法
        myCat.testInstanceMethod();             //调用 myCat 对象的实例方法
        Animal myAnimal2 = new Animal();
        myAnimal2.testInstanceMethod();         //调用 myAnimal2 对象的实现方法
    }
}
```

在上面的示例中，类 Cat 覆盖了类 Animal 中的实例方法并隐藏了 Animal 中的类方法。在 main 方法中，创建一个 Cat 类的实例对象并调用 Animal 和 myAnimal、Cat 的类方法 testClassMethod()和 myAnimal 的实例方法 testInstanceMethod()。

④ 编译并运行 ExtendsDemo4.java 程序，输出结果如下。

```
Animal 中的类方法.
Animal 中的类方法.
Cat 中的类方法.
Cat 中的实例方法.
Cat 中的实例方法.
Animal 中的实例方法.
```

可以看出，得到调用的隐藏方法的版本是超类中的方法，而得到调用的覆盖方法的版本是子类中的方法。也就是说，当调用隐藏的方法时，根据引用变量声明的类型来决定调用超类还是子类中相应的静态方法。如果声明的对象类型是超类的，即使它实际指向的是子类，也调用超类的静态方法。而当调用覆盖的方法时，则根据其实际指向对象的类型来决定调用超类还是子类中相应的实例方法。

7.4.4 方法覆盖和方法隐藏后的修饰符

在子类中被重写的方法，其访问权限允许大于但不允许小于被其覆盖的方法。也就是说，子类中重写方法的访问控制修饰符的作用域，要大于超类中被覆盖的方法的访问控制修饰符的作用域。例如，超类中一个受保护的实例方法(protected)在子类中可以是公共的（public）的，但不可以是私有的（private）。

不允许将超类中的一个实例方法在子类中改变为类方法，否则将会产生一个编译时错误。反之亦然。换句话说，如果一个方法在超类中是 static 方法，那么在子类中也必须是 static 方法；如果一个方法在超类中是实例方法，那么在子类中也必须是实例方法。

7.4.5 方法覆盖和方法隐藏总结

表 7.1 总结了当在子类中定义一个与超类中的方法具有相同签名的方法时，会发生的情况。

表 7.1 子类方法与超类方法具有相同的签名

	超类实例方法	超类静态方法
子类实例方法	覆盖	产生编译时错误
子类静态方法	产生编译时错误	隐藏

需要注意的是，在一个子类当中，可以重载从超类继承过来的方法。这些重载的方法既不隐藏也不覆盖超类的方法，它们是新的方法，对于子类来说是唯一的。

【例 7.6】方法重载示例。

```
// 父类 A
class A{
    //表示问候的公共方法
    public void sayHello( ){
        System.out.println("Hello,everyone! ");
    }
}
// A 的子类 B
class B extends A{
    //重载的 sayHello 方法
    public void sayHello(String helloStr){
        System.out.println(helloStr);
    }
}
//主程序 ExtendsDemo5
public class ExtendsDemo5{
    public static void main(String[ ] args){
        B b = new B( );
```

```
        b.sayHello( );                          //调用父类中的 sayHello 方法
        b.sayHello("Hello! " );                 //调用子类中重载的 sayHello 方法
    }
}
```

编译并运行上面的程序，输出结果如下。

```
Hello,everyone!
Hello!
```

在这个程序中，父类 A 和子类 B 中有两个同名的方法 sayHello()，但子类中的 sayHello()方法带有一个字符串类型的参数，而父类中的 sayHello()方法不带参数，所以子类中的 sayHello()方法重载了从父类中继承过来的 sayHello()方法。这样，在子类中就有两个可用的 sayHello()方法，一个不带参数，一个带有一个 String 类型的参数。

当通过 B 的对象实例 b 调用其 sayHello()方法时，根据是否带有参数来决定调用哪一个。当调用不带参数的 sayHello()方法时，会调用从父类 A 继承过来的 sayHello()方法；当调用带参数的 sayHello()方法时，会调用在子类 B 中重载的 sayHello()方法。

7.5　隐藏超类中的字段

如果一个子类中与其父类有同名的字段，即使这些字段的数据类型不同，那么也说，子类中的字段隐藏了超类的字段。在子类中，不能通过简单的名称来引用父类中的这个字段，必须通过关键字 super 访问它。例如，在类 A 中定义了一个 String 类型的变量 name，在其子类 B 中也定义了一个同名的变量，数据类型也为 String。

【例 7.7】隐藏超类中的字段示例。

```
// 父类 A
class A{
    public String name = "张小明";
}
// A 的子类 B
class B extends A{
    public String name = "赵红丽";
    public void self( ){
        System.out.println("我的名字是：" + name);
                                        //此处调用子类 B 中声明的字段 name
    }
}
// 主程序 ExtendsDemo6
public class ExtendsDemo6{
    public static void main(String[ ] args){
        B b = new B( );                 //创建类 B 的实例对象
        b.self( );                      //调用对象的 self 方法
    }
}
```

编译并运行上面的程序，输出结果为：

```
我的名字是：赵红丽
```

可以看出，子类中的 name 字段隐藏了父类中的 name 字段。一般而言，不推荐使用隐藏字段，因为这有可能使代码难以阅读。

7.6 使用 super 关键字访问超类中的成员

如果子类中的一个方法覆盖了超类中的一个方法，那么要想在子类中可以访问到父类中被覆盖的方法，可以在子类中通过使用关键字 super，来调用超类中被覆盖的方法。还可以在子类中通过使用 super 来指向被隐藏的字段。

7.6.1 调用超类中被覆盖的方法

如果超类中的某个方法被子类所覆盖，那么如果想从子类中访问这个方法，就要使用 super 关键字，告诉编译，其后调用的方法为超类中的方法。下面的示例程序理解如何使用 super 关键字调用超类中被覆盖的方法。

【例 7.8】使用 super 关键字访问超类中的成员方法。

（1）创建超类 Superclass，其中包含方法 printMethod()。

```
public class Superclass {
    public void printMethod() {
        System.out.println("在 Superclass 中的输出。");
    }
}
```

（2）创建 Superclass 类的子类，名为 Subclass，它覆盖了方法 printMethod()。

```
public class Subclass extends Superclass {
    //重写超类 Superclass 中的 printMethod 方法
    public void printMethod() {
        super.printMethod();              //使用 super 关键字调用超类中的方法
        System.out.println("在 Subclass 中的输出。");
    }
}
```

（3）编写一个包含 main 方法的测试程序 ExtendsDemo7.java。

```
public class ExtendsDemo7{
    public static void main(String[] args) {
        Subclass s = new Subclass();                    //创建子类的一个实例对象
        s.printMethod();                                //调用子类中重写的方法
    }
}
```

在子类 Subclass 中，简单的方法名 printMethod()代表 Subclass 中声明的方法，它重写了 Superclass 中的该方法。因此，要调用从 Superclass 中继承过来的 printMethod()方法，在 Subclass 中必须使用一个限定名称，使用 super 关键字，如示例中所示。编译并执行 Subclass，输出结果如下。

```
在 Superclass 中的输出。
在 Subclass 中的输出。
```

7.6.2 访问超类中被隐藏的字段

在例 7.5 中，当在子类中覆盖了父类中同名的字段以后，就无法通过子类调用父类中被覆盖的同名字段。这时如果需要通过子类调用父类中被覆盖的同名的字段，可以使用



super 关键字，后跟圆点运算符及要调用的字段。其语法形式如下所示。

```
super.字段名
```

下面修改 7.5 节中的示例程序，使用 super 关键字访问父类中被隐藏的字段。

【例 7.9】使用 super 关键字访问父类中被隐藏的字段。

```
// 父类 A
class A{
    public String name = "张小明";
}
// A 的子类 B
class B extends A{
    public String name = "赵红丽";       //与父类中同名的字段
    public void self( ){
        System.out.println("我的名字是：" + name);
        System.out.println("我原来的名字是：" + super.name);
                                          //使用 super 访问超类中被隐藏的字段
    }
}
// 主程序 ExtendsDemo8
public class ExtendsDemo8{
    public static void main(String[ ] args){
        B b = new B( );                   //创建子类 B 的实例对象
        b.self( );                        //调用其 self 方法
    }
}
```

编译并运行上面的程序，输出结果如下。

```
我的名字是：赵红丽
我原来的名字是：张小明
```



7.7　使用 super 关键字调用超类的构造器

前面讲过，子类不继承其父类的构造器。因此，如果要初始化父类中的字段，可以在子类的构造器中，通过关键字 super 调用父类的构造器。对超类的构造器的调用必须放在子类构造器的第一行。本节就详细介绍如何使用 super 关键字调用超类的构造器。

7.7.1　调用超类的无参构造器

调用一个超类构造器需要使用 super 关键字，当使用 super()时，超类的无参构造器会被调用。其语法如下所示。

```
super();
```

【例 7.10】调用超类的无参构造器示例。

```
// 父类 A
class A{
    A(){                                  //父类构造器
        System.out.println("这里是超类 A 的构造器。");
    }
}
```

```
// A 的子类 B
class B extends A{
    B(){                                          //子类构造器
        super();                                  //首先调用其父类构造器
        System.out.println("这里是子类 B 的构造器。");
    }
}
// 主程序 ExtendsDemo9.java
public class ExtendsDemo9{
   public static void main(String[] args) {
        B b = new B();                            //创建子类 B 实例对象，调用其构造器
    }
}
```

编译并运行 ExtendsDemo9.java 程序，输出结果如下所示。

```
这里是超类 A 的构造器。
这里是子类 B 的构造器。
```

实际上，如果一个构造器没有显式地调用一个超类的构造器，那么 Java 编译器会自动地插入一个对超类无参构造器的调用。例如，将上面示例中子类 B 中的“super();”语句去掉，如下所示。

```
// A 的子类 B
class B extends A{
    B(){
        System.out.println("这里是子类 B 的构造器。");
    }
}
```

输出结果不变。虽然在子类 B 的构造器中没有显式地调用超类的构造器，但是 Java 编译器自动地插入了对超类 A 的无参构造器的调用。如果超类没有一个无参构造器，就会产生编译时错误。例如，将上例中父类 A 中的无参构造器改为带参数的构造器，如下所示。

```
//父类 A
class A{
    A(int a){
        System.out.println("这里是超类 A 的带有参数的构造器。");
    }
}
//A 的子类 B
class B extends A{
    B(){
        System.out.println("这里是子类 B 的构造器。");
    }
}
//主程序 ExtendsDemo9.java
public class ExtendsDemo9{
   public static void main(String[] args) {
        B b = new B();
    }
}
```

编译 Test.java 程序，会出现如下的编译时错误提示信息。

```
Test.java:10: 找不到符号
```



```
符号:  构造函数 A()
位置:  类 A
       B(){
          ^
1 错误
```

可以看出，由于超类 A 中没有无参构造器（当一个类中显式定义一个构造器时，Java 就不会为其创建默认的无参构造器了），所以在编译时，在 B 类的构造器中，首先调用超类 A 的无参构造器，而超类 A 并没有无参的构造器，这样就会产生一个编译时错误。

Object 肯定有一个无参构造器，所以如果一个类没有明确地指明其超类，那么默认其超类是 Object，在这种情况下不会产生什么问题。

7.7.2　调用超类的带参构造器

如果要完成对父类成员的一些初始化工作，那么使用无参的 super()方法就不可以了。这时，需要使用带有参数的 super()方法，其语法形式如下所示。

```
super(参数列表);
```

下面，先回忆一下，在本章开始的“继承”一节中，给出了 AutoCar 类及其子类 Car 类的代码，如下所示。

```
public class AutoCar{
    //定义三个私有字段，代表 AutoCar 对象的三个属性
    private String brand;                    //表示汽车的品牌
    private float exhaust;                   //表示汽车的排气量
    private int speed;                       //表示汽车的速度
    //定义一个带参的构造器，用来初始化字段
    public AutoCar(String brand,float exhaust,int speed){
        this.brand = brand;
        this.exhaust = exhaust;
        this.speed = speed;
    }
    //获得属性 brand 的值
    public String getBrand(){
        return brand;
    }
    //获得属性 exhaust 的值
    public float getExhaust(){
        return exhaust;
    }
    //获得属性 speed 的值
    public int getSpeed(){
        return speed;
    }
    //设置属性 speed 的值
    public void setSpeed(int newSpeed){
        speed = newSpeed;
    }
}
```

在创建 AutoCar 类的子类 Car 类时，在 Car 类的构造器中带有三个参数。在构造器中，首先调用了其超类 AutoCar 的构造器，然后添加它自己的初始化代码，如下所示。

```
    public class Car extends AutoCar{
        //Car 子类中增加一个字段，代表载重量
        public int weight;
        //Car 子类有一个构造器
        public Car(int startWeight,String originalBrand,float originalExhaust,
int startSpeed){
            super(originalBrand,originalExhaust,startSpeed); //调用超类的构造器
            weight = startWeight;                            //初始化子类的字段
        }
        //Car 类增加一个方法
        public void setWeight(int newWeight){
            weight = newWeight;
        }
    }
```

使用“super（参数列表）”，超类的带有相匹配的参数列表的构造器会被调用。因为构造器不能被继承，那么对从父类中继承过来的字段的初始化，必须通过调用超类的带参构造器来完成，如上面的程序所示。因此，在子类的带参构造器中，参数列表中包括父类构造器要用到的参数。

7.7.3 构造器链

如果一个子类的构造器显式或隐式地调用其超类的一个构造器，而其超类会再显式或隐式地调用超类的超类的构造器，那么将会存在一个完整的构造器的连环调用，一路调用到 Object 的构造器。这称为“构造器链”。当存在一个很长的类的继承链时，在使用时要考虑到这一点。

【例 7.11】构造器链示例。

```
//父类 A，其继承自 Object 类
class A{
    A(){                                          //超类的无参构造器
        System.out.println("这里是超类 A 的构造器。");
    }
}
//A 的子类 B
class B extends A{
    B(){                                          //子类 B 的无参构造器
        System.out.println("这里是子类 B 的构造器。");
    }
}
//B 的子类 C
class C extends B{
    C(){                                          //子类 C 的无参构造器
        System.out.println("这里是子类 C 的构造器。");
    }
}
//主程序
public class ExtendsDemo10{
   public static void main(String[] args) {
        C c = new C();          //创建子类 C 的实例对象，会依次调用父类无参构造器
    }
}
```



在这个示例中，类 B 继承自类 A，而类 C 又继承自类 B。当在 main()方法中使用类 C 的构造器构造一个 C 类的对象实例时，会先调用其父类 B 的构造器，然后才执行其自身构造器中的其他语句。同样的道理，在调用 B 的构造器时，也会先调用其父类 A 的构造器，依此类推，在类 A 的构造器中也是先调用其父类 Object 类的构造器。

因此在执行这个程序时，最先执行的输出语句是类 A 的构造器中输出语句，即最先输出“这里是超类 A 的构造器。”，其次是类 B 的构造器中的输出语句得到执行：“这里是子类 B 的构造器。”。最后输出的语句是类 C 中的“这里是子类 C 的构造器。”编译并运行上述程序，输出结果如下。

```
这里是超类 A 的构造器。
这里是子类 B 的构造器。
这里是子类 C 的构造器。
```

可以看出，这个程序示例的构造器链如图 7.2 所示。

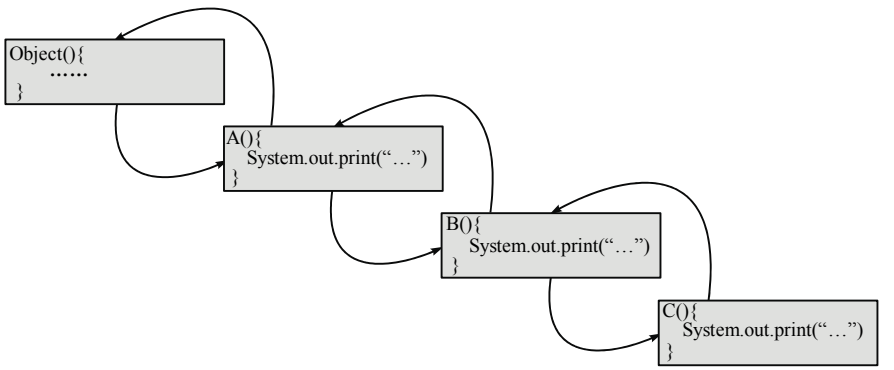


图 7.2　Java 中的构造器链



7.8　Object 类

在前面的章节中，在很多地方都提到了 Object 类。Object 类是很特别的一个类，位于 java.lang 包中，处于 Java 类层级树的顶端。Java 中的所有类，都是 Object 类的直接或间接子孙类。Object 类为 Java 中的所有类提供了最基本的属性和方法，是 Java 中所有类的高度概括。

7.8.1　作为超类的 Object 类

在 Java 中，每一个类都是 Object 类的直接或间接子类。程序中所使用或编写的每一个类都继承了 Object 的实例方法。通常不会使用这些方法，但是如果要使用的话，就需要使用特定的代码来覆盖它们。本节要讨论的从 Object 继承过来的方法如下所示。

```
//创建并返回本对象的一个复制
protected Object clone() throws CloneNotSupportedException
//判断一个对象是否与另一个对象相等
public boolean equals(Object obj)
//当垃圾收集器决定内存中不再有对一个对象的引用时，垃圾收集器在对象上调用该方法
protected void finalize() throws Throwable
//返回一个对象的运行时的类
public final Class getClass()
```

```
//返回对象的哈希值
public int hashCode()
//返回对象的一个字符串表示
public String toString()
```

另外，Object 还有 notify、notifyAll 和 wait 等方法，主要用在同步程序的独立的线程活动中，这将在后面的相关章节中讲解。类似这样的方法总共有 5 个，如下所示。

```
public final void notify()
public final void notifyAll()
public final void wait()
public final void wait(long timeout)
public final void wait(long timeout, int nanos)
```

7.8.2 Object 类的对象克隆方法 clone()

如果一个类或其超类实现了接口 Cloneable，那么可以使用接口中声明的 clone()方法从一个已经存在的对象创建一个克隆的新对象。要创建一个克隆对象，语法格式如下。

```
aCloneableObject.clone();
```

其中，aCloneableObject 是一个实现了 Clone()方法的对象。在执行这个方法时，会先检查对象是否实现了 Cloneable 接口。如果这个对象没有实现该接口，那么就会抛出一个 CloneNotSupportedException 异常。

如果一个对象实现了 Cloneable 接口，那么调用其 clone()方法会创建一个新的对象，并将新的对象的成员变量初始化为与原始对象相应成员变量相同的值，这个新的对象与原对象具有相同的类型。

使一个类具有可克隆能力的一个最简单的方法，是在类的声明中加入 implements Cloneable，然后就可以在这个类的对象上调用 clone()方法。

对于有的类来说，Object 的 clone()方法工作良好。但是，如果一个对象包含有对一个外部对象的引用，例如一个名为 OjbExternal 的外部对象，那么就需要重写 clone()方法以得到正确的行为。否则，在一个对象中对 ObjExternal 所做的改变，在这个对象的克隆对象中也是可见的。这意味着，原始对象和它的克隆对象是不相互独立的。

要断开它们之间的联系，必须覆盖 clone()方法，这样它会克隆对象及外部对象 ObjExternal。然后原始对象引用一个 ObjExternal 而克隆对象引用一个克隆的 ObjExternal，它们引用的不是同一个对象了，这样原始对象和克隆对象就真正地相互独立了。

7.8.3 Object 类的对象比较方法 equals()

equals()方法比较两个对象的相等性。如果两个对象相等，就返回 true，否则，返回 false。在 Object 类中所提供的 equals()方法使用恒等运算符（==）来判断两个对象是否相等。对于原始数据类型，使用这种方式来判断的结果是正确的。但是对于引用类型的对象，则不一定正确。Object 所提供的 equals()方法测试对象的引用是否相等，也就是说，两个进行比较的对象是否是同一个对象。

但是在实际应用中，经常需要进行比较的是两个对象的内容是否相同，而不是判断两个对象的地址是否相同，即两个对象是否等价。要测试两个对象是否等价，就必须重写 equals()方法。



【例 7.12】重写 equals()方法的 Book 类。

```
class Book {
    private String ISBN;                    //声明一个私有的字符串变量，代表书号
    Book(String isbn){                      //带有一个参数的构造器
            this.ISBN = isbn;               //初始化字段 ISBN
    }
    public String getISBN(){                //获取 ISBN 字段值的方法
            return this.ISBN;
    }
    public boolean equals(Object obj) { //两个对象进行比较的方法
        if (obj instanceof Book)            //如果比较对象是 Book 类的实例
           return ISBN.equals(((Book)obj).getISBN());
                                            //根据两本书的书号是否相同，返回比较结果
        else
           return false;
    }
}
```

再添加一个 ExtendsDemo11 类，测试 Book 类的两个实例的相等性，如下所示。

```
public class ExtendsDemo11{
    public static void main(String[] args) {
         Book firstBook  = new Book("0201914670");       //创建第一本书的实例
         Book secondBook = new Book("0201914670");       //创建第二本书的实例
         if (firstBook.equals(secondBook)) {             //对两本书进行比较
             System.out.println("两个对象相等.");
         } else {
             System.out.println("两个对象不相等.");
         }
    }
}
```

编译并运行此程序，输出结果如下。

```
两个对象相等。
```

虽然 firstBook 和 secondBook 分别引用不同的对象，但结果是相等的，因为这两个对象包含有相同的 ISBN 书号。

当相等运算符不适合自己的类时，应该重写 equals()方法。需要注意的是，如果重写了 equals()方法，就必须同时重写 hashCode()方法。

7.8.4　Object 类的对象清除方法 finalize()

Object 类提供了一个回调方法 finalize()，当一个对象成为“垃圾”时，既在内存中不存在任何对该对象的引用时，该对象的 finalize()方法可能会被调用。Object 类的 finalize()方法实现什么也不做。程序员可以在自己的类中重写 finalize()方法来做一些清除工作，例如释放资源。

finalize()方法也可能被系统自动地调用，但是什么时候调用，甚至是否会被调用，是不确定的。因此，程序员不应该依赖此方法进行清理工作。

7.8.5　Object 类的获取对象类信息的方法 getClass()

不能覆盖 getClass()方法。方法 getClass()返回一个 Class 对象。Class 对象有一些方法，

可以用来获取一些类的信息，如类名（getSimpleName()）、它的超类（getSuperclass()），以及它所实现的接口（getInterfaces()）。例如，下面的代码获取并显示一个对象的类的名称。

```
void printClassName(Object obj) {
   System.out.println("对象所属的类是" + obj.getClass().getSimpleName());
}
```

Class 类位于 java.lang 包中，它拥有许多的方法，数量超过 50 个。例如，可以测试看看某个类是否是一个注解（isAnnotation()）、一个接口（isInterface()）或一个枚举类型（isEnum()）。还可以看看对象的字段都有哪些（getFields()）、方法有哪些（getMethods()）等。

7.8.6 Object 类的获得对象内存地址的方法 hashCode()

由 hashCode()方法返回的值是对象的哈希值，它是对象以十六进制表示的内存地址。根据定义，如果两个对象是相等的，那么它们的哈希值也必须相等。

如果用户在自己写的类中覆盖了 equals()方法，就改变了两个对象被换算的方式，并且 Object 的 hashCode()方法的实现就不再有效了。因此，如果重写了 equals()方法，就必须同时也重写 hashCode()方法。

7.8.7 Object 类的对象字符串表示方法 toString()

Object 类的 toString()方法返回一个对象的字符串表示，这对于程序调试是非常有用的。一个对象的字符串表示完全依赖于对象本身，所以要在自己的类中覆盖 toString()方法。

可以连同 System.out.println()语句一起使用 toString()方法，以显示一个对象的文本表示，例如 Book 的一个实例：

```
System.out.println(firstBook.toString());
```

如果适当地重写了 toString()方法，那么在使用上述语句时将会输出一些有用的信息，以帮助用户更好地理解类，例如：

```
ISBN:020914670:精通 Java 程序设计
```

在自己编写的类中，应该总是考虑去覆盖 toString()方法。

7.9 编写 final 类和 final 方法

在实际开发中，如果一个类非常重要，程序员不想它被其他人改变其属性和方法，也就是说，不希望其子类改变它的某些方法，或者根本就不想被继承，那么可以使用 final 关键字来声明 final 方法或 final 类。

7.9.1 使用 final 关键字声明 final 方法

使用 final 关键字，可以将一个类的一些方法或所有方法都声明为 final 类型的，在一个方法声明中使用 final 关键字以说明此方法不能被子类覆盖。例如：

```
protected final void doXXX(){……}
```

在 Object 类中就有许多 final 方法。如果一个方法有一个不应该被改变的实现并且此



方法严格与对象状态保持一致的状态，那么需要将此方法声明为 final 类型的。例如，在一个国际象棋相关的程序中，规则是执白先行，这个规则是固定的，不应该被改变，所以可以将程序中执先者的方法 getFirstPlayer 方法设置为 final 类型的：

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK };          //声明一个枚举类型
    ...
    //使用 final 关键字声明不可被覆盖的方法
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...
}
```

从构造器中调用的方法通常应该被声明为 final 型的。如果一个构造器调用一个非 final 型的方法，那么其子类可能会重新定义这个方法，从而带来意想不到的结果。

7.9.2　使用 final 关键字声明 final 类

还可以声明一个 final 类，这意味着这个类是不能被继承的，也就是说，它不能有子类。这样定义有特定的用处，如有时为了安全性，将一些重要的类声明为 final 类。例如，Java 平台中的 String 类和 System 类都非常重要，有许多重要的实现，因此都被定义为 final 类。声明 final 类的语法格式如下。

```
final class 类名称{
    //类体
}
```

【例 7.13】创建一个名称为 FinalClassDemo 的 final 类。

```
public final class FinalClassDemo{
    private String message = "这是一个 final 类";
    private String enable = "它不能被继承，所以不可能有子类。";
    public static void main(String[] args){
        FinalClassDemo demo = new FinalClassDemo( );   //创建类的实例对象
        System.out.println(demo.message);  //输出实例对象的 message 字段的值
        System.out.println(demo.enable);   //输出实例对象的 enable 字段的值
    }
}
```

在上面的程序中，定义了一个 final 类型 FinalClassDemo。当类被声明为 final 时，这个类中的所有字段和方法都是 final 类型的，并且此类不能被继承。

7.10　小结

本章主要学习了类的继承。这是面向对象语言的三大特征之一。本章学习的重点是继承中方法的覆盖和方法的重写、super 关键字的使用及构造器链。读者在学习本章内容时对 super 关键字的使用及构造器链的调用要细心体会，这是本意学习的难点。

通过本章的学习，读者应该对类的继承有深刻的理解，并能正确而熟练地通过继承创建新的类。在第 8 章中，将向读者介绍 Java 语言的另一个重要的概念：接口。

第8章 接口

在软件工程中，经常将一个大的项目分由不同的开发小组共同开发，各个小组的开发工作是并行的。在很多时候，对于不同小组的程序员来说，清楚地说明各自的软件如何相互作用，并就此达成统一“约定”是很重要的。这种情况下，每一小组都可以在对其他小组怎么编写代码一无所知的情况下，就能编写自己的代码，并且各自小组编写的代码能很好地协同工作。一般而言，接口就是这样的“约定”。



8.1 理解接口

在 Java 中，不同的对象有时需要具有相似的行为。例如，电视机的开关按钮，DVD 机的开关按钮，饮水机的开关按钮等。按下开关按钮，打开或关闭电视机、DVD 机或饮水机。这些对象（电视机、DVD、饮水机等）具有相似的行为：按下按钮，就打开或关闭电源。这些都是它们对外的接口。

8.1.1 一般接口的概念

电视机、DVD、饮水机等这些对象，它们内部实现打开或关闭的原理可能是很不相同的，但对于用户来说，它们的操作是相同的，只需要“按下”开关按钮，就可以打开或关闭，这就降低了用户使用新的家用电器的难度。对于家用电器制造商来说，要使自己的产品更容易被用户所接受，那么它所生产的家用电器都必须实现统一的“约定”，既都有相似的开关按钮，暴露给用户相似的操作。

也就是说，各种家用电器都要实现同一接口，在接口中对家用电器的电源开关有统一的约定，而具体的操作由各个电器自己去实现，这就保证了各个家用电器所提供的开关行为更加统一，而用户在不必知道其开关原理的情况下，就能正确开关这些家用电器。

设想一下下面这个场景：在未来的社会中，由计算机控制的遥控汽车在城市的街道中穿梭，运输乘客，而没有一个人类操作员。汽车制造商用 Java 语言编写操纵汽车的软件，该软件具有各种功能：停止、启动、加速、左转等。另外的工业团体（如无线电制导设备制造商）生产相关的计算机系统，用来接收 GPS（全球定位卫星）定位数据和无线传输的交通信息，并使用这些信息来驾驶这些汽车。



汽车制造商必须公布一个工业标准的接口，详细地说明哪些方法可以被调用，以控制汽车移动（包括任何制造商的任何汽车）。然后无线电制造商就可以编写相关的软件，在其中调用在接口中描述的方法，以对汽车发出控制命令。两个工业团体都不需要知道其他团体的软件是如何实现的。实际上，每一个团体都认为他自己的软件是高度地私有的，并拥有在任何时候对它进行修改的权限，只要该软件继续使用公开发布的接口。

在这里，使用接口对汽车可以使用的方法做了统一的声明，而不同的汽车制造商针对自己不同类型的汽车，相应地去实现这些方法。而对无线电制导设备制造商来说，他只需知道，凡是实现了工业标准接口的汽车，都具有接口中所说明的方法：停止、启动、加速、左转等。因此，在制导设备制造商所制造的相关计算机系统中，不必知道各种型号的汽车是怎么实现这些功能的，它只知道调用了相关的方法就能完成相关的功能。

因此，只要汽车制造商坚持使用工业标准的接口，那么即使它修改了汽车停止、启动、加速和转向的实现原理，也不影响制导设备制造商。汽车制造商和制导设备制造商完全可以独立地展开各自的工作。

8.1.2　Java 中的接口

在 Java 程序设计语言中，接口是一个引用类型，与类相似，所以可以在程序中定义并使用一个接口类型的变量。

在接口中只能包含有常量、方法签名和嵌套类型，它没有构造器。在接口中声明的方法，没有方法体。因为接口没有构造器，所以它不能被实例化，它只能被类所实现或者被另外的接口所继承。

8.1.3　在 Java 中定义接口

在 Java 中，接口也属于引用类型。定义接口与创建一个新的类很类似，但需要使用关键字 interface，后跟接口的名称。接口的名称同样要遵循 Java 标识符命名规则。定义接口的语法格式如下。

```
public interface OperateCar {
   // 如果有常量的话，在这里声明
   // 方法声明
   int turn(Direction direction,// Direction 是一个枚举类型，值为 RIGHT 和 LEFT
            double radius, double startSpeed, double endSpeed);
   int changeLanes(Direction direction, double startSpeed, double endSpeed);
   int signalTurn(Direction direction, boolean signalOn);
   int getRadarFront(double distanceToCar, double speedOfCar);
   int getRadarRear(double distanceToCar, double speedOfCar);
        ......
   // 更多的方法声明
}
```

注意接口中的方法签名后面没有花括号，以分号结尾。

8.1.4　在 Java 中使用接口

要想使用接口，就需要编写一个实现接口的类。当一个类实现一个接口时，它必须为

接口中声明的每一个方法提供一个方法体，即必须实现接口中的所有方法。例如，下面是一个操纵宝马车的类，它实现了 OperateCar 接口：

```
public class OperateBMW760i implements OperateCar {
   // 实现 OperateCar 接口中的方法。例如：
   int signalTurn(Direction direction, boolean signalOn) {
      //打开宝马车的左转指示灯
      //关闭宝马车的左转指示灯
      //打开宝马车的右转指示灯
      //关闭宝马车的右转指示灯
   }
   // 其他需要的成员 – 例如，辅助类
}
```

在上面自动控制的汽车例子中，由汽车制造商将实现这些接口。例如，雪佛兰汽车的实现将会完全不同于丰田汽车的实现，然而，两个汽车制造商都遵循同样的接口。而无线制导设备制造商，作为接口的客户，将负责构建相关的计算机系统，用来使用 GPS 定位数据、数字街道地图和传输数据驱动汽车。在其实现过程中，制导系统将会调用接口的方法：转向、改变路线、刹车、加速等。不论是雪佛兰汽车，还是丰田汽车，因为它们实现了相同的接口，所以都有这些方法以供调用，并完成相应的功能。

8.1.5 作为 API 的接口

在上面的自动控制的汽车示例中，展示了一个被作为工业标准的应用程序接口（API）使用的接口。作为 API 使用的接口还普遍使用在商业软件生产中。典型地，一个软件公司会开发并销售包含有复杂方法的一个软件包，而另外一家公司在它们自己的软件产品中使用这些方法。

例如，A 公司开发一个含有数字图像处理方法的软件包，然后将此软件包销售给制作终端用户图片程序的 B 公司。A 公司编写它们自己的类以实现一个接口，而将接口公开给它的客户 B。然后 B 公司根据接口中定义的方法签名和返回类型，调用相应的图像处理方法。当 A 公司的 API 对它的客户 B 公司公开时，它对 API 的实现实际上是隐藏的。事实上，A 公司可能会在以后的日子里修改 API 的实现，升级其软件包。但是，只要新的软件包继续实现原始接口，都不会影响到它的客户 B 公司对 API 的使用。

8.1.6 接口和多继承

在 Java 语言中，接口有另外一个非常重要的作用。因为 Java 中不允许多继承，一个类只能继承自一个父类，因此有时这种单继承并不能反映现实世界的某些现象。例如，未来的某个时候，出现了一种超级汽车，可以在水、陆、空三种环境下行驶。但是它只能从最初的 AutoCars 类继承，只继承了陆地行驶的方法，这时只能过单继承的方式不能满足创建此超级汽车的目的。

因此在 Java 中规定，一个类虽然只能继承自一个父类，但它可以实现多个接口。这时，要想创建能水、陆、空都能行驶的汽车，除了从 AutoCars 类继承陆地行驶的方法之外，还可以实现带有水中行驶和空中行驶方法的接口，那么它就具有了水中行驶和空中行驶的方法。当实现多个接口时，对象可以同时具有多个类型：



- 自身所属类的类型。
- 其所实现的所有接口的类型。

这意味着，如果声明一个接口类型的变量（接口是引用类型），那么它的值可以引用任何实现了该接口的类的任何实例对象。



8.2　定义接口

有时候，程序员需要自己定义要使用的接口或者用来分发的接口。定义接口与定义类很相似，包括接口的声明和接口体的实现。在接口体中，含有对接口所包含的所有的方法的方法声明。接口所含有的方法声明后面紧跟一个分号，而不是花括号，因为一个接口不提供对它里面所声明的方法的实现。在一个接口中声明的所有方法都隐含是 public 的，所以 public 修饰符可以被省略。

8.2.1　接口声明

接口声明由修饰符、关键字 interface、接口名称、用逗号分隔的父接口列表（如果有的话）和接口体组成。声明接口的语法形式如下。

```
[修饰符] interface 接口名称 [extends 父接口名称列表]{
    // 接口体
}
```

说明如下。

- 修饰符：可选，用于指定接口的访问权限，可选值为 public。如果省略，则使用默认的访问权限，即只能在当前的软件包中使用。换句话说，声明接口时，关键字 interface 前面要么是修饰符 public，要么什么都没有，而不能使用 protected 或 private 关键字。
- interface：必选，定义接口的关键字。
- 接口名称：必选，用于指定接口的名称。接口名必须是合法的 Java 标识符。一般情况下，要求接口名称的首字母大写。
- extends 父接口名称列表：可选，用于指定该接口继承自哪个父接口。当使用 extends 关键字时，父接口名称为必选参数。接口可以是多继承的，即一个接口可以有任意多个父接口。在接口的声明中，包括其所有父接口的一个列表，用逗号分隔。

例如，下面声明一个接口 GroupedInterface，该接口继承自三个父接口：

```
public interface GroupedInterface extends Interface1,Interface2,Interface3 {
    // 接口体
}
```

在上面的声明中，public 访问控制符指出该接口可以被任何包中的任何类所使用。如果没有指定接口是 public 的，那么接口将只能被与其定义在同一个包中的类所访问。

8.2.2　接口体

在接口声明后面的大括号括起来的部分，为接口的接口体。接口体由两部分组成：常量声明和方法声明。其语法格式如下所示。

```
[修饰符] interface 接口名称 [extends 父接口名称列表]{
    // 常量声明
    [public] [static] [final] 常量名称;
    // 方法声明
    [public] [abstract] 返回类型 方法签名;
}
```

说明如下。

- 常量声明：接口中可以包含有常量声明，也可以不包含，它不是必需的，根据需要而定。如果有常量声明的话，默认是 public static final 类型的，也就是说，接口中的所有字段都隐含地具有 public、static 和 final 属性。所以可以省略常量声明的修饰符 public、static 和 final，其作用都是一样的。
- 方法声明：接口中的方法只有返回类型和方法签名，而没有方法体。接口中的方法都具有 public 和 abstract 属性，所以在声明方法时，可以省略前面的修饰符 public 和 abstract，作用都是一样的。也就是说，即使声明方法时前面不使用修饰符，该方法也隐含地是 public 和 abstract 的。

下面定义的接口包含有对接口中常量的声明和方法的声明。

```
public interface GroupedInterface extends Interface1,Interface2,Interface3 {
    // 常量声明
    double E = 2.718282;  // 自然对数的基数
    //上面的变量声明等价于 public static final double E=2.718282 语句
    // 方法声明
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

在接口的接口体中，含有对接口所包含的所有方法的声明。接口所含有的方法声明后面紧跟一个分号，而不是花括号，因为一个接口不提供对它里面所声明的方法的实现。在一个接口中声明的所有方法都隐含是 public 的，所以 public 修饰符可以被省略。

8.3 实现接口

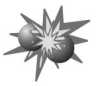
接口的主要作用，是声明共同的常量或方法，用来为不同的类提供不同的实现，但这些类仍然可以保持同样的对外接口。接口可以被类实现，也可以被其他接口继承。在类中实现接口需要使用关键字 implements。

8.3.1 实现接口的语法

要声明一个实现接口的类，需要在类的声明中使用 implements 短语。一个类可以实现多个接口，所以 implements 关键字后面要跟一个被类实现的接口列表，用逗号分隔符分隔。在类中实现接口的语法格式为：

```
[修饰符] class 类名称 [extends 父类名称] implements 接口列表{
    // 类体
    // 在类中，要实现所有接口中声明的方法
}
```

按惯例，implements 短语跟在 extends 短语后面（如果有 extends 短语的话）。



8.3.2　接口实例：Relatable

下面考虑一个比较不同对象大小的接口实例 Relatable。在接口 Relatable 中，声明了一个 isLargerThan 方法，其作用是和另外一个实现了 Relatable 接口类型的对象进行比较。Relatable 接口的声明如下所示。

```
public interface Relatable {
    public int isLargerThan(Relatable other);
}
```

调用 isLargerThan 方法的对象和方法参数中的 other 对象必须是同一个类的实例。当调用此方法的对象比方法参数中的 other 对象大、相等或小时，此方法分别返回 1、0、-1 。

如果两个相似的对象想比较相互的大小，不管这些对象是什么，实例化它们的类都应该实现 Relatable 接口。

理论上，任何类都可以实现 Relatable 接口，只要从这些类实例化的对象有办法比较相对“大小”。例如，对于字符串，可以是比较字符的个数；对于图书，可以是比较页数；对于学生，可以是年龄等。对于二维几何对象，对面积进行比较是较合适的，而对于三维几何对象，对体积进行比较是较合适的。所有诸如此类的类都可以实现 isLargerThan()方法。如果知道一个类实现了 Relatable 接口，那么就知道可以对这个类所实例化的对象进行大小的比较。

8.3.3　实现接口 Relatable

在 6.4.2 节“创建对象”中，讲过一个代表矩形的 Rectangle 类。代表两个矩形的 Rectangle 类的对象通常需要比较它们面积的大小。下面是对其重新编写以实现接口 Relatable 后的形式。

【例 8.1】实现了 Relatable 接口的类及其接口方法调用。其实现步骤如下。

① 编写一个 RectanglePlus 类，并实现 Relatable 接口。

```
public class RectanglePlus implements Relatable {
    public int width = 0;           //声明接口中的属性
    public int height = 0;
    public Point origin;            //声明对象属性
    // 四个构造器
    public RectanglePlus() {        //无参构造器
        origin = new Point(0, 0);   //使用默认值（0，0）初始化矩形左上角坐标
    }
    public RectanglePlus(Point p) { //带有一个 Point 类型参数的构造器
        origin = p;                 //使用已知的点来初始化矩形左上角坐标
    }
    public RectanglePlus(int w, int h) {    //带有两个参数的构造器
        origin = new Point(0, 0);   //使用默认值（0，0）初始化矩形左上角坐标
        width = w;                          //使用已知的值 w 来初始化矩形的宽度
        height = h;                         //使用已知的值 h 来初始化矩形的高度
    }
    public RectanglePlus(Point p, int w, int h) {   //带有三个参数的构造器
        origin = p;                 //使用已知的点来初始化矩形左上角坐标
        width = w;                  //使用已知的值 w 来初始化矩形的宽度
        height = h;                 //使用已知的值 h 来初始化矩形的高度
    }
    // 移动矩形的方法
```

```
    public void move(int x, int y) {
        origin.x = x;                    //改变矩形对象左上角的坐标值
        origin.y = y;
    }
    // 计算矩形面积的方法
    public int getArea() {
        return width * height;
    }
    // 实现接口 Relatable 中声明的方法
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect = (RectanglePlus)other;    //对象类型转换
        if (this.getArea() < otherRect.getArea())
                                         //如果当前对象的面积小于参数对象的面积
            return -1;                   //返回-1
        else if (this.getArea() > otherRect.getArea())
                                         //如果当前对象的面积大于参数对象的面积
            return 1;                    //返回 1
        else
            return 0;                    //如果当前对象与参数对象的面积相等，返回 0
    }
}
```

因为类 RectanglePlus 实现了 Relatable 接口，所以任何两个 RectanglePlus 对象都可以比较面积的大小。

② 另外编写一个含有 main()方法的 ImlDemo1.java 应用程序，使用 isLargerThan 方法比较大小：

```
public class ImlDemo1{
    public static void main(String[] args){
        RectanglePlus rect1 = new RectanglePlus(3,5);
                             //创建实现了 Relatable 接口的 RectanglePlus 对象
        RectanglePlus rect2 = new RectanglePlus(2,7);
                             //创建实现了 Relatable 接口的 RectanglePlus 对象
        int result = rect1.isLargerThan(rect2);
                             //调用接口中的方法进行对象的比较
        switch(result){      //根据比较的结果，输出相应的信息
            case -1:
                System.out.println("矩形 rect1 比矩形 rect2 小。");
                break;
            case 1:
                System.out.println("矩形 rect1 比矩形 rect2 大。");
                break;
            default:
                System.out.println("矩形 rect1 与矩形 rect2 一样大小。");
        }
    }
}
```

在上面的 main()方法中，创建两个实现了相同的 Relatable 接口的类的实例，并调用相应的接口方法进行比较。

8.3.4 编译并运行带有接口的实例

上一节的程序共包含有 4 个类型，其中有三个类（Point 类、RectanglePlus 类、主类



ImlDemo1），有一个接口 Relatable。要想让这个实例运行起来，可以有两种形式：

- 将 4 个类型（3 个类和 1 个接口）分别保存为单独的源文件，文件名和类名或接口名相同（Point.java、RectanglePlus.java、Relatable.java、Test.java）。4 个源文件放在同一个目录中。然后通过命令行窗口编译主程序 ImlDemo1.java，编译器会自动地在当前目录下寻找相关的类进行编译，最后运行 ImlDemo1 的类文件。
- 将 4 个类型（3 个类和 1 个接口）放在一个源文件中，并将除 Test 类之外的其他类和接口声明前的 public 修饰符去掉（在一个源文件中，只能有一个类是 public 的，并且类名与源文件名相同，一般都将含有 main 方法的类设为 public 的）。然后编译并运行 ImlDemo1.java 程序。

8.3.5　实现多个接口时的常量和方法冲突问题

在 Java 中，每个类只能实现单重继承，而实现接口时，则可以实现多个接口，每个接口使用逗号分隔符分隔。这时就有可能出现常量或方法名冲突的情况。例如，一个类实现两个接口，两个接口中都声明了相同名称的常量或相同名称的方法，那么在实现这两个接口的类中，引用常量或实现方法时，就不明确是哪个接口中的。

要解决这类冲突问题时，如果是常量冲突，则需要在类中使用全限定名（接口名称.常量名称）明确指定常量所属的接口。如果是方法冲突，则只需要实现一个方法就可以了。

【例 8.2】在下面这个示例程序中，定义了两个接口，并且在这两个接口中都声明了一个同名的常量和一个同名的方法。然后再定义一个同时实现这两个接口的类。实现步骤如下。

（1）创建 Calculate 接口，在该接口中声明一个常量和两个方法：

```
//位于文件 Calculate.java 中
public interface Calculate {
    float PI = 3.14f;                    //定义一个用于表示圆周率的常量 PI
    float getArea( );                    //定义一个用于计算面积的方法
    float getGirth( );                   //定义一个用于计算周长的方法
}
```

（2）创建 Shape 接口，在该接口中声明一个常量和两个方法：

```
//位于文件 Shape.java 中
public interface Shape{
    float PI = 3.14159f;                 //定义一个用于表示圆周率的常量 PI
    float getArea( );                    //定义一个用于计算面积的方法
    void draw( );                        //定义一个绘制图形的方法
}
```

（3）创建一个 Circle 类，该类同时实现 Calculate 接口和 Shape 接口：

```
//位于文件 Circle.java 中
public class Circle implements Calculate,Shape {
    //声明一个私有变量 radius，代表圆的半径
    private float radius;
    //构造器，用于初始化半径 radius
    public Circle(float r){
        radius = r;                              //将圆的半径初始化为参数 r 的值
    }
    //实现计算圆面积的方法，该方法两个接口中都有，只需实现一个即可
```

```
    public float getArea( ){
        float area = Calculate.PI*radius*radius;
                        //计算圆的面积，使用全限定名指定 Calculate 接口中的常量
        return area;
    }
    //实现计算圆周长的方法
    public float getGirth ( ){
        float circuGirth = 2*Shape.PI*radius ;
                        //计算圆的面积，使用全限定名指定 Shape 接口中的常量
        return circuGirth;
    }
    //实现绘制图形的方法
    public void draw( ){
        System.out.println("在这里绘制了一个圆！");
    }
}
```

（4）创建一个含有 main()方法的主测试程序：

```
//位于文件 ImlDemo2.java 中
public class ImlDemo2{
    public static void main(String[ ] args){
        Circle circle = new Circle(7);          //创建 Circle 的对象
        float area = circle.getArea( );
                        //调用 circle 对象的 getArea 方法，获得该对象的面板值
        System.out.println("圆的面积为：" + area);
        float girth = circle.getGirth( );
                        //调用 circle 对象的 getGirth 方法，获得该对象的周长值
        System.out.println("圆的周长为：" + girth);
        circle.draw( );     //调用 circle 对象的 draw 方法，绘制一个圆
    }
}
```

（5）编译并运行 ImlDemo2.java 程序，输出结果如图 8.1 所示。

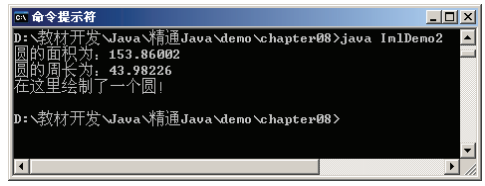


图 8.1　多接口程序运行结果

在这个程序中，接口 Calculate 和接口 Shape 都声明有 getArea()方法和变量 PI，而类 Circle 实现了这两个接口，所以为了避免变量冲突，在引用 PI 时，使用了全限定名 Calculate.PI；为了避免方法冲突，只实现一个 getArea()方法。

8.4　将接口作为类型使用

当定义一个新的接口时，就是定义一个新的引用数据类型。在任何能够使用其他数据类型名称的地方，也可以使用接口名称。如同其他数据类型一样，可以定义一个接口类型的引用变量。

如果定义一个接口类型的引用变量，那么，赋给这个变量的任何对象就必须是实现这个接口的类的一个实例。例如，在下面的示例方法中，用来在两个对象中查找最大的那个对象。对于这其中的任何一个对象来说，不管是 object1 还是 object2，都必须是实现了



Relatable 接口的类的实例对象：

```
public Object findLargest(Object object1, Object object2) {
     Relatable obj1 = (Relatable)object1; //将参数 object1 转换为 Relatable 对象
     Relatable obj2 = (Relatable)object2; //将参数 object2 转换为 Relatable 对象
     if ( (obj1).isLargerThan(obj2) > 0) //对两个对象进行比较
        return object1;
     else
        return object2;
}
```

在上面的代码中，通过将 object1 转换为一个 Relatable 接口类型，它就可以调用 isLargerThan 方法。同样地，实现了接口 Relatable 的类的实例对象也都可以使用下面的方法进行比较：

```
public Object findSmallest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;    //将参数 object1 转换为 Relatable 对象
   Relatable obj2 = (Relatable)object2; //将参数 object2 转换为 Relatable 对象
   if ( (obj1).isLargerThan(obj2) < 0)   //对两个对象进行比较
      return object1;
   else
      return object2;
}
public boolean isEqual(Object object1, Object object2) {
                                          //定义两个对象进行比较的方法
   Relatable obj1 = (Relatable)object1; //将参数 object1 转换为 Relatable 对象
   Relatable obj2 = (Relatable)object2; //将参数 object2 转换为 Relatable 对象
   if ( (obj1).isLargerThan(obj2) == 0) //判断两个对象是否相等
      return true;
   else
      return false;
}
```

这些方法可以用于任何“有关联的”对象，不管这些对象的类是怎么继承的。当一个对象实现 Relatable 接口时，它既可以是自身类（或超类）的类型，也可以是一个 Relatable 类型。这就提供了一些多继承的好处，这些对象可以拥有它们超类的行为，同时也可以拥有它们所实现的接口的行为。



8.5　改写接口

一个接口被声明以后，有可能因为某种原因，需要改变接口中声明的方法（如增加新的方法、改变方法体的实现等）。例如，假设某个程序员 A 编写了一个名为 DoIt 的接口，如下所示。

```
public interface DoIt {
    void doSomething(int i, double x);              //接口中声明的方法
    int doSomethingElse(String s);                  //接口中声明的方法
}
```

这个接口可以提供给其他的程序员使用。其他的程序员在自己的类中实现这个接口。假设稍后的某个时间，程序员 A 想要给 DoIt 增加第三个方法，那么这个接口现在变成如

下的样子：

```
public interface DoIt {
    void doSomething(int i, double x);
    int doSomethingElse(String s);
    boolean didItWork(int i, double x, String s);      //新增加的方法声明
}
```

但是问题出现了。因为如果这样做了，那么所有已经实现了旧的 DoIt 接口的类将不能正常使用，因为它们只实现了旧的接口中的两个方法（doSomething()和 doSomethingElse()），而对于修改过的接口来说，少实现了第三个新增的方法 didItWork()，这等于没有再实现这个接口。

要避免出现这种情况，最好的办法是努力预测接口的所有功能并从一开始就进行详细的说明。但这在实际开发中是很难做到的，因此，最合适的做法是，保持原来的接口不动，从原来的接口派生一个新的接口，在新的接口中增加第三个方法。例如，程序员 A 应该创建一个继承自 DoIt 接口的 DoItPlus 接口：

```
public interface DoItPlus extends DoIt {
    boolean didItWork(int i, double x, String s);
                                                    //在新的接口中声明要增加的方法
}
```

现在其他的程序员可以选择继续使用旧的接口 DoIt，或者升级到新的接口 DoItPlus。这就避免了因为直接修改接口，而导致的依赖这个接口的程序不能正常使用的这种情况。

 8.6 小结

本章主要介绍了接口的概念、接口的声明及接口的实现。在学习本章内容时，读者重点理解接口的概念和学习如何实现接口。在将接口作为类型使用上，读者可能会感到难以理解，需用心体会。下一章向读者介绍两个特殊的类：抽象类和嵌套类。

第9章　抽象类与嵌套类

Java 语言中，还包含一些特殊的类定义方法，例如抽象类、嵌套类。它们都属于 Java 类的高级特性，都有特殊的用途。抽象类用来提供更高级的类型抽象，而嵌套类则提供更加紧密的封装特性。本章为读者详细介绍抽象类和嵌套类的概念与使用方法。



9.1　抽象类和抽象方法

在面向对象的程序设计语言中，类是有层次和继承关系的。子类总是比其超类更加具体。而程序员在程序开发过程中，很多时候需要构造一系列的类及其继承关系，这时候，通常会将类层次中共有的特性抽取出来，创建包含这些共有特性的抽象类，并由抽象类派生出更加具体、有更多实现的子类或后代类，形成一个完整的类的层次体系。

9.1.1　抽象方法

如果一个方法被声明但是没有被实现（即没有花括号，没有方法体，声明后面直接就是分号），那么该方法被称为“抽象方法”。例如下面的代码所示。

```
abstract void moveTo(double deltaX, double deltaY);
```

方法 moveTo()即为抽象方法。抽象方法只有方法的声明，而没有方法的实现，用关键字 abstract 进行修饰。其语法格式如下。

```
abstract <方法返回值类型> 方法名称(参数列表);
```

除了关键字 abstract，其他声明与普通方法声明的要求相同。在“接口”一章中讲到过，接口中的方法都是只有声明，没有实现的方法，与这里的抽象方法很类似。实际上，在一个接口中的所有方法，隐含地都是抽象的，因此接口的方法可以不使用 abstract 修饰符。当然也可以使用，只不过不是必需的。

9.1.2　抽象类

如果一个类被声明为 abstract，那么称这是一个“抽象类”。抽象类可能含有抽象方法，也可能没有，但抽象类中可以含有非抽象的方法。抽象类不能被实例化，不能使用 new 运

算符创建抽象类的实例对象，但是抽象类可以派生子类。在定义抽象类时，要在关键字 class 前面加上关键字 abstract。其语法格式如下。

```
abstract class 类名称{
    类体
}
```

如果一个类包含有抽象方法，那么这个类必须被声明为 abstract，如下所示。

```
public abstract class GraphicObject{
    //声明字段
    //声明非抽象的方法
    abstract void draw( );      //声明抽象方法
}
```

当从一个抽象类派生子类时，派生出来的子类通常提供对其超类中的所有抽象方法的实现。如果子类中没有实现其超类中所有的抽象方法，那么这个子类也必须被声明为 abstract（抽象的）。

9.1.3 抽象类与接口对比

与接口不同，抽象类可以包含有非 static 和 final 的字段，并且抽象类也可以包含有实现了的方法。这样的抽象类与接口很相似，但抽象类提供部分方法的实现，其余的让子类来完成实现。如果一个抽象类只包含抽象方法的声明，那么应该将其声明为一个接口。

多接口可以被类层次中的任何类所实现，无论这些类之间是否有任何形式的关系。例如，Comparable 或 Cloneable 接口。抽象类最经常用于被子类化，并共享部分实现。在一个单独的抽象类中，提供大部分子类的共同点（即抽象类已经实现的部分），但是还有一些不同点（抽象方法）暂时只声明不实现，留给不同的子类根据自己的要求去具体实现。

9.1.4 抽象类实例

假设在一个使用面向对象技术的绘图程序中，可以绘制圆、矩形、直线、贝塞儿曲线，以及其他图形对象。这些对象有很多共同点，如都有一定的状态（位置、方位、线条颜色、填充颜色等）和行为（移动、旋转、缩放、绘制等）。有的状态和行为对所有的绘图对象来说都是相同的，例如：位置、填充颜色和移动。其他的要求有不同的实现，例如：缩放或绘制。

所有的绘图对象必须知道如何缩放或绘制它们自身；而缩放和绘制只是在不同的对象上实现的方法不一样。这非常适用于一个抽象类。可以利用这些相似性，将所有的绘图对象都声明为继承自同一抽象的父对象。例如，定义一个抽象的父对象类 GraphicObject，它们之间的继承关系如图 9.1 所示。

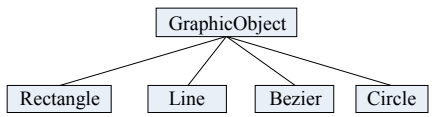


图 9.1 类 Rectangle、Line、Bezier 和 Circle 都继承自 GraphicObject

首先，声明一个抽象类 GraphicObject，在这个类中，提供被所有的子类所全部共享的成员变量和方法，如当前位置和 moveTo()方法。在 GraphicObject 中还声明一些抽象方法，如 draw()或 resize()，这些方法需要被子类实现，但是实现的途径又各自不同。声明的抽象类 GraphicObject 如下所示。

```
abstract class GraphicObject {
```



```
    //声明被所有的子类所共享的成员变量
    int x, y;
    ...
    //声明已经实现的成员方法，可以被所有的子类所共享
    void moveTo(int newX, int newY) {
        ...
    }
    //声明抽象方法，由各个子类去具体实现
    abstract void draw();
    abstract void resize();
}
```

每一个 GraphicObject 类的非抽象子类，如代表圆的 Circle 和代表矩形的 Rectangle，必须实现 draw()和 resize()方法：

```
//声明代表圆的 Circle 类，是抽象类 GraphcObject 的子类
class Circle extends GraphicObject {
    //在子类中实现父类中声明的抽象方法，提供具体的实现
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
//声明代表矩形的 Rectangle 类，是抽象类 GraphcObject 的子类
class Rectangle extends GraphicObject {
    //在子类中实现父类中声明的抽象方法，提供具体的实现
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

很显然，虽然代表圆的 Circle 类和代表矩形的 Rectangle 类都有 draw()方法用来绘制自身，都有 resize()方法来改变自身的大小，但它们实现的方式肯定是不同的。这样就在抽象类 GraphicObject 中提供了统一的对外接口，而在子类中去具体实现。这也是多态的一种。

9.1.5　抽象类实现接口

在“接口”一章讲到，如果一个类实现了一个接口，那么它必须实现接口中所有的方法。但是也有可能定义一个不必实现全部接口方法的类，这需要将该类声明为抽象的(abstract)来实现。例如：

```
abstract class X implements Y {
    // 实现除了一个方法之外 Y 中所有其他方法
}
class XX extends X {
    // 实现 Y 中剩下的未被实现的方法
}
```

在上面这种情况下，类 X 必须是抽象的（声明为 abstract 类），因为它没有全部地实现接口 Y，但是类 XX 实际上全部实现了 Y。

9.1.6 抽象类的类成员

一个抽象类也可以有静态字段和静态方法（static 字段和 static 方法）。与其他类一样，可以直接使用抽象类名来引用这些静态成员，例如：

```
abstract class AbstractClass{
    static void say(){                    //定义的类方法(静态方法)
        System.out.println("hello");
    };
}
public class Test{
   public static void main(String[] args) {
      AbstractClass.say();                //可以直接通过抽象类名调用其静态方法
   }
}
```

需要注意的是，修饰符 static 和 abstract 不能一起使用。如果一个方法声明为静态的，就不能声明为抽象的；如果声明为抽象的，就不能声明为静态的。

9.2 嵌套类

在 Java 程序设计语言中，允许在一个类如 ClassA 中定义另一个类如 ClassB，这样类 ClassB 被称为“嵌套类”，而包含 ClassB 的类 ClassA 称为“外部类”。例如，在下面的代码示例中，定义了外部类 OuterClass，并且在外部类 OuterClass 中定义了嵌套类 NestedClass：

```
class OuterClass {
   ...
   class NestedClass {
      ...
   }
}
```

嵌套类分为两种类别：静态的和非静态的。声明为 static 的嵌套类称为“静态嵌套类”。非静态嵌套的类被称为“内部类”。如下所示。

```
class OuterClass {
   ...
   //静态嵌套类
   static class StaticNestedClass {
      ...
   }
   //内部类
   class InnerClass {
      ...
   }
}
```

包围嵌套类的类称为外部类，如这里的 OuterClass。一个嵌套类是其外部类的成员。非静态嵌套类（内部类）可以访问其外部类的其他成员，即使是声明的私有（private）成员。静态嵌套类不能访问其外部类的其他成员。作为“外部类”的一个成员，一个嵌套的类可以被声明为 private、public、protected 或者包级私有的（package private）（而外部类只能被声明为 public 或包级私有）。



9.2.1　为什么要使用嵌套类

通常情况下，使用嵌套类的情形并不多，但在编写事件响应的代码时，相对使用嵌套类的时候较多。使用嵌套类有多种原因，包括：

- 这是将只用在同一个地方的类进行逻辑上的分组的一种方法。如果一个类只对另一个类有用，那么将其逻辑地嵌入另一个类并使两个类紧密地结合在一起。嵌套这样的“辅助类”使得它们所在的包更加简化和有效。
- 增强了封装性。例如两个顶级类 A 和 B，B 需要访问 A 中被声明为 private 的成员。通过将类 B 隐藏在类 A 中，A 的成员可以被声明为私有的，同时 B 也可以访问它们。另外，B 本身对外部世界是隐藏的。
- 嵌套类能使代码可读性和可维护性更强。在顶级类中嵌套较小的类使得代码最接近它被使用的地方。

9.2.2　静态嵌套类

与类方法和类变量一样，静态嵌套类与它的外部类相关联。并且像静态类方法一样，一个静态嵌套类不能直接引用其外部类的实例变量或实例方法，只能通过一个对象引用使用它们。

一个静态嵌套类就像任何其他顶级类一样与它的外部类（以及其他类）的实例成员交互。从效果上来说，一个静态嵌套类的行为表现得就像一个顶级类，只不过为了包装的便利，将其嵌套进了另一个顶级类中。使用其外部类的名称访问静态内部类，如下所示。

```
OuterClass.StaticNestedClass
```

例如，要创建一个静态嵌套类的对象，使用下面所示的语法：

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```

9.2.3　内部类

就像实例方法和变量，一个内部类与包围它的外部类的实例相关联，并且可以直接访问那个对象的方法和字段。此外，因为内部类与一个实例相关联，所以在内部类中不能定义任何静态成员。一个内部类的实例对象存在于其外部类的实例对象中。考虑下面的代码：

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

一个内部类的实例只能存在于外部类的实例中，并可以直接访问包围它的实例的方法和字段，包括私有方法和私有字段。图 9.2 说明了这种概念。

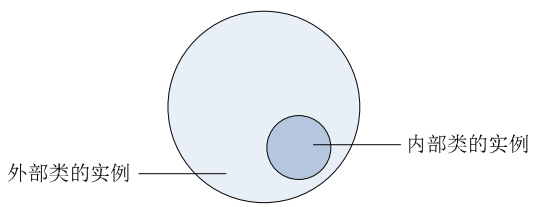


图 9.2　内部类存在于外部类的实例中

要实例化一个内部类，必须首先实例化其外部类。然后使用下面的语法创建位

于外部对象中的内部对象：

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass() ;
```

一个嵌套类可以访问它的封装类的所有私有成员，包括字段和方法。因此，被一个子类所继承的一个 public 或 protected 嵌套类可以间接访问超类的所有私有成员。另外，还有两个特定种类的内部类：局部类和匿名类（也叫匿名内部类）。

9.2.4 内部类实例

在下面的实例中，将创建一个数组，使用整数填充该数组，然后按升序只输出数组偶数索引的元素值。在下面的 DataStructure 类中，有下列几个组成部分：

- DataStructure 外部类，它包含添加一个整数到数组的方法和输出数组中偶数索引处值的方法。
- InnerEvenIterator 内部类，它与标准的 Java 迭代器相似。迭代器被用于遍历一个数据结构，并有测试最后一个元素、获得当前元素及移动到下一个元素的方法。
- 实例化一个 DataStructure 对象（ds）的 main()方法，在 main()方法中使用 ds 将整数（如 0，1，2，3 等）填充到数组 arrayOfInts 中，然后调用 printEven()方法输出数组 arrayOfInts 中的偶数索引的值。

【例 9.1】创建一个数组，使用整数填充该数组，然后按升序只输出数组偶数索引值。代码如下。

```
public class DataStructure {
    //创建一个数组
    private final static int SIZE = 15;              //声明常量，表示数组大小
    private int[] arrayOfInts = new int[SIZE];       //创建一个 int 类型的数组
    public DataStructure() {
        //使用升序排列的整数值填充数组
        for (int i = 0; i < SIZE; i++) {
            arrayOfInts[i] = i;
        }
    }
    public void printEven() {                        //输出数组偶数索引的值
        InnerEvenIterator iterator = this.new InnerEvenIterator();
                                                     //创建嵌套类的实例
        while (iterator.hasNext()) {
            System.out.println(iterator.getNext() + " ");
        }
    }
    //实现 Iterator 模式的内部类
    private class InnerEvenIterator {
        //从头开始遍历数组
        private int next = 0;
        public boolean hasNext() {
            //检查当前元素是否是数组中的最后一个元素
            return (next <= SIZE - 1);
        }
        public int getNext() {
            //记录数组的一个偶数索引的值
            int retValue = arrayOfInts[next];
```



```
            //获得下一个偶数元素
            next += 2;
            return retValue;
        }
    }
    public static void main(String s[]) {
        //使用整数值填充数组并只输出偶数索引处的值
        DataStructure ds = new DataStructure();
        ds.printEven();
    }
}
```

编译并运行程序 DataStructure.java，输出结果为：

```
0 2 4 6 8 10 12 14
```

注意，在上面的示例程序中，内部类 InnerEvenIterator 类直接引用了外部类 DataStructure 对象的 arrayOfInts 实例变量。

内部类可以被用于实现辅助类，就像上例中那样。Java 的事件处理机制广泛地使用了内部类，因此如果要学会处理用户接口事件，就需要知道怎样使用内部类。

9.2.5　局部内部类和匿名内部类

还有另外两种特殊的内部类：局部内部类和匿名内部类。可以在一个方法的方法体中声明一个内部类。这样的类称为“局部内部类”。还可以在一个方法的方法体中声明一个没有命名的内部类，这样的类称为“匿名内部类”。

9.2.6　内部类的修饰符

凡是可以用于外部类的其他成员的修饰符，都可以用于内部类。例如，可以使用访问控制修饰符 private、public 和 protected 来限制对内部类的访问，就像限制对其他的类的成员进行访问控制限制一样。

9.3　小结

本章介绍了两种特殊的类型：抽象类和嵌套类。抽象类是实现多态的一种途径，嵌套类主要应用在事件监听器的构建上。其中抽象类和接口的异同是读者学习的一个重点和难点。嵌套类的定义也是学习中容易迷惑的地方。建议读者在学习到这部分内容时，如果暂时不能理解的话，不妨先放一放。等学过后面的事件监听机制以后，再回来学习这部分内容，相信就容易理解了。

下一章将向读者介绍 Java 语言的新特性之一：泛型。

第10章 泛型

泛型是 Java 中引入的新特性。通过在程序中使用泛型，可以提高程序代码的复用性，减少数据的类型转换，从而可以提高代码的运行效率，并使代码更健壮。在 Java 集合框架中大量地使用了泛型，泛型也主要用在这些集合类中。本章主要学习泛型的语法和术语。



10.1 泛型简介

在任何正式的软件项目中，bug 都会存在，这是无法更改的事实。仔细地计划、设计和测试可以帮助降低代码中 bug 的影响，但是不管怎样去做，不管在什么地方，总是不能完全消除 bug 的存在。随着在程序中引入新的特性，以及代码量增加、程序复杂性增长，遇到的问题会越来越突出。而通过使用泛型，可以协助程序员进行代码检测，增加代码的稳定性。

10.1.1 为什么要使用泛型

在程序中，出现的 bug 一般有两类。一类是编译时的 bug，它们比另外的 bug 更加容易检测出来。例如，编译时的 bug 能够立即提示哪里出现错误了；可以使用编译器的错误信息来判断出问题是什么，并进行处理，在程序运行之前就予以纠正。例如语句没有以分号结尾，或者方法的定义少了右花括号等。这都可以在编译是由虚拟机检测出来，程序员根据提示信息，当时就可以改正。

但是有的 bug 在编译时是检测不出来的，而在运行期间，当满足一定条件时，就会使程序出现问题。这种运行时的 bug 更加让人迷惑：它们并不总是立即表现出来，但是当它们表现出来时，它出现的位置可能已经离问题的根源很远了。Java 中引入泛型，通过在编译时进行更多的 bug 检测，从而增强代码的稳定性。

10.1.2 一个简单的 Box 类

首先从设计一个非泛型的 Box 类开始，这个类代表一个对象的“盒子”，它操作任何类型的对象：把对象放入盒子或从盒子中取出对象。它只需要提供两个方法：add()方法和 get()方法。Add()方法用来添加一个对象到一个盒子；而 get()方法从盒子中重新获得该对象。Box 类的代码如下。

```
public class Box {
    private Object object;           //声明一个 Object 对象
    public void add(Object object) {//定义 add 方法,对本实例的 object 对象赋值
        this.object = object;
    }
    public Object get( ) {           //定义 get 方法,返回本实例的 object 对象
        return object;
    }
}
```

因为 Box 类的方法接收或返回 Object，因此可以传入任何类型的参数（原始数据类型除外）。但是，如果这个“盒子”Box 只用来存放和获取一个指定类型的对象，既将其需要将包含的类型限定为某些指定的类型（例如 Integer），那么唯一的选择是在帮助文档中详细说明这种要求（或者像本例一样，在注释中说明），当然编译器对此限定是一无所知的：

```
public class BoxDemo1 {
    public static void main(String[ ] args) {
        // 只能将 Integer 对象放入此箱子中!
        Box integerBox = new Box();
        integerBox.add(new Integer(10));  //调用add方法,为Box对象添加一个整型对象
        Integer someInteger = (Integer)integerBox.get();    //对象类型转换
        System.out.println(someInteger);
    }
}
```

在上面的 BoxDemo1 中，程序创建一个 Integer 对象，将它传递给 add()方法，然后通过 get()方法重新获得该对象，并赋给同样的 Integer 对象 someInteger。最后输出对象的值 10 到标准输出。

在“继承”一章讲过，将类型从 Object 转换到 Integer 是正确的，因为 Object 类型的引用实际上是 Interger 对象实例。并且这在注释中已经做了详细的说明，程序员在使用这个类前，要注意到这个注释并按要求去做。但是编译器对此一无所知，它只信赖正确的转换。所以，如果一个粗心的程序员传递进一个错误类型的对象，例如 String，编译器并不知道这是错误的。请看下面的程序：

```
public class BoxDemo2 {
    public static void main(String[] args) {
        //只能将 Integer 对象放入此箱子中!
        Box integerBox = new Box();
        // 设想这是一个大的程序的一部分，被一个程序员所修改.
        integerBox.add("10");            // 注意现在的类型是 String
        // ... 假设这里是另外一个程序员所编写的
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
}
```

在上面的 BoxDemo2 中，将一个整数 10 作为一个 String 进行存储。这在程序中经常遇到，例如一个程序的 GUI 所收集的用户输入的值，一般是作为字符串类型的。

但是，因为这个程序可能是两个程序员合作完成的，或者虽然是一个程序员编写的，但在后面的时候忘记了，那么在从 Box 中获取放入的对象时，想当然地认为应该是 Integer 类型的，从而将一个指向 String 类型的 Object 引用变量向 Integer 的类型转换。这很明显

是一个 bug，但是，因为代码仍然能够编译通过，所以程序员并不知道有错误存在，直到运行的时候，程序崩溃并抛出一个 ClassCastException（类型转换异常）：

```
Exception in thread "main"
        java.lang.ClassCastException:
           java.lang.String cannot be cast to java.lang.Integer
           at BoxDemo2.main(BoxDemo2.java:6)
```

上面异常信息的意思是：java.lang.String 不能够转换为 java.lang.Integer 类型。很显然，如果能在一开始就检测到这种 bug 的存在，就可以避免到运行的时候才出现问题。将问题解决在它发生前总比发生了以后再解决要好。所以，如果 Box 类一开始就使用泛型被用心地设计，这样的错误将会被编译器所捕获，而不会在运行时引起程序崩溃。



10.2 泛型类型

下面使用泛型来更新上一节的 Box 类。泛型类型实际上是通过给类或接口增加类型参数（Type Parameters）实现的。类型参数指明，泛型类型可以用在多种数据类型上，在具体使用时，可以根据实际指定的类型来决定。

10.2.1 Box 类的泛型版本

声明类的泛型版本与声明类的一般版本的语法格式很类似，只有稍微的不同。例如，在第 6 章中讲过，声明一般类的语法格式为：

```
[修饰符] class 类名 [extends 父类名] [implements 接口 1 名,接口 2 名,…]{
    类体
}
```

而声明泛型类时，只需要在类名后面跟上一个类型参数即可，其语法格式为：

```
[修饰符] class 类名<类型参数> [extends 父类名] [implements 接口 1 名,接口 2 名, 。。。]{
    类体
}
```

将上一节的 Box 类改写为泛型类，首先通过将声明类的代码“public class Box”改为“public class Box<T>”以创建一个“泛型类型声明”；这里引入了一个“类型变量”，称为 T，可以在该类（同样地，泛型也可以被用于接口）中的任何地方使用这些类型变量，在一定程度上将它们当做已知的类型。

实际上，“类型变量”T 和一般意义上的变量非常类似，只要把 T 想象为一个代表指定类别的变量，它的“变量值”是传递进来的任何类型：即可以是任何类类型，任何接口类型，或者甚至另外一个类型变量。但是它不能是任何的原始数据类型。

在这里，假设 T 是 Box 类的一个“类型参数”。这里的 T 就像是类 Box 的参数，用来接收使用时传进来的实际类型：

```
// Box 类的泛型版本.
public class Box<T> {
    private T t;          // T 代表"类型"
    public void add(T t) {
        this.t = t;
    }
    public T get() {
```



```
        return t;
    }
}
```

在上面的代码中，将所有出现 Object 的地方用 T 来代替。要在程序中应用这个具有泛型特性的 Box 类时，必须执行一个“泛型类型调用”，指明实际的具体类型，即在每个类型变量处分别用一个实际的具体类型替代使用具体的值替换掉 T，例如使用 Integer：

```
Box<Integer> integerBox;
```

可以将泛型类型调用想象成与一个普通的方法调用相类似，但是给方法传递的是一个实参，而这里是传递一个“类型实参”（在本例中是 Integer）到 Box 类。与其他任何变量声明一样，这个代码并不真正地创建一个新的 Box 对象。它简单地声明 integerBox 将会持有一个对“Integer 类型的 Box”（Box<Integer>）的引用。

一个泛型类型调用通常被看做是“参数化类型”。要实例化这个类，照常使用 new 关键字，但是将<Integer>放在类名和圆括号之间：

```
integerBox = new Box<Integer>();
```

或者，可以将整个语句放在一行，例如：

```
Box<Integer> integerBox = new Box<Integer>();
```

一旦 integerBox 被初始化，就可以随便调用它的 get()方法，而不再需要使用类型转换，如下面的示例所示。

```
public class BoxDemo3 {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        Integer someInteger = integerBox.get();      // 不需要进行类型转换!
        System.out.println(someInteger);
    }
}
```

此外，如果添加一个不匹配的类型到 Box 对象中，例如将 add()方法的参数改为 String，编译将不会通过，提示错误信息：

```
    Test.java:14: 无法将 Test<java.lang.Integer> 中的 add(java.lang.Integer)
应用于 (java.lang.String)
        integerBox.add("10");
                   ^
    1 错误
```

正确地理解类型变量实际上并不是类型本身，这很重要。在上面的示例中，在文件系统中的任何地方都找不到 T.java 或 T.class。而且，T 不是 Box 类的名称的一部分。实际上，在编译期间，所有的泛型信息将完全地被移除，在文件系统中只留下 Box.class。这在稍后的“类型擦除”一节再详细解释。

一个泛型类型还可能有多个类型参数，就像一个方法可以有多个参数一样。其语法格式如下。

```
[修饰符] class 类名<类型参数 1，类型参数 2,。。。，类型参数 n>
                          [extends 父类名] [implements 接口 1 名,接口 2 名,…]{
    类体
}
```

但是每个参数在它的声明类或接口中必须是唯一的。例如，声明 Box<T,T>，将会在第二个出现 T 的地方生成一个错误，但是如果声明为 Box<T,U>，就是合法的。

10.2.2 类型参数命名惯例

按惯例，类型参数命名为单个的大写字母。这与已经知道的变量命名规则不太相同。这样命名的原因是：如果不这样命名，将很难区分在一个类型变量和一个变通类或接口名称之间的不同。最普遍使用的类型参数名称如下：

- E——元素（被 Java 集合框架所广泛地使用）。
- K——键。
- N——数字。
- T——类型。
- V——值。
- S，U，V 等——第二、第三、第四个类型。

 10.3 泛型方法和泛型构造器

类型参数还可以在方法和构造器签名中声明，用来创建“泛型方法”和“泛型构造器”。这与声明一个泛型类型相似，但是类型参数的作用域被限制在它被声明的方法或构造器中。请看下面这个示例代码：

```
// 这个版本介绍一个泛型方法.
public class Box2<T> {
    private T t;                    //声明私有的泛型类型 T 的变量 t
    public void add(T t) {          //声明泛型方法，该方法在调用时要传入 T 类型的参数
        this.t = t;
    }
    public T get() {                //声明泛型方法，该方法被调用时返回一个 T 类型数据
        return t;
    }
    // 定义一个泛型方法
    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
    public static void main(String[] args) {
        Box2<Integer> integerBox = new Box2<Integer>();
                                            //创建一个 Integer 类型的 Box2 对象
        integerBox.add(new Integer(10));//调用其 add 方法，添加一个 Integer 对象
        integerBox.inspect("some text");//调用 inspect 方法
    }
}
```

在上面的代码中，增加了一个名为 inspect()的泛型方法。在这个方法中，定义一个名为 U 的类型参数。这个方法接收一个对象并输出它的类型到标准输出。作为对比，还输出了 T 的类型。为了方便执行，这个类现在还有一个 main()方法，这样就可以被作为应用程序运行。上面这个程序的输出结果如下。

```
T: java.lang.Integer
U: java.lang.String
```



通过传递进来不同的类型，输出结果也相应地改变。泛型方法的实际的用处可能会像下面这样，定义一个静态方法，该方法将对一个单个条目的引用填充到多个 Box 对象中。

```
public static <U> void fillBoxes(U u, List<Box<U>> boxes) {
    for (Box<U> box : boxes) {
        box.add(u);
    }
}
```

要使用上面这个方法，首先要有下面这样的代码：

```
Crayon red = …;
List<Box<Crayon>> crayonBoxes = …;
```

调用 fillBoxes 的完整语法如下。

```
Box.<Crayon>fillBoxes(red,crayonBoxes);
```

在这里，显式地提供了作为 U 使用的类型 Crayon，不过一般不这样写，可以省略而由编译器推断所需要的类型。例如：

```
Box.fillBoxes(red, crayonBoxes);        //编译器推断 U 是 Crayon
```

这种特性，称为“类型推断”，允许像调用一个普通方法一样调用一个泛型方法，而不必指定一个类型。



10.4　限定的类型参数

有时，程序员想将允许被传递的类型参数的类型限制到一定的范围之内。例如，操纵数字的方法可能只想接收 Number（数字类）或其子类的实例。这就是“限定类型参数”的目的。

要声明一个限定的类型参数，通过列出类型参数的名字，后跟 extends 关键字，再后面跟它的“上限”（在本例中是 Number）来实现。注意，在这个上下文中，extends 是通常意义上的用法，它既意味着“继承”（如在类中一样），也意味着“实现”（如在接口中一样）。请看下面的示例：

```
// 这个版本引入一个限定类型参数
public class Box3<T> {
    private T t;                    //声明私有的泛型类型 T 的变量 t
    public void add(T t) {          //声明泛型方法，该方法在调用时要传入 T 类型的参数
        this.t = t;
    }
    public T get() {                //声明泛型方法，该方法被调用时返回一个 T 类型数据
        return t;
    }
    // 定义一个泛型方法，将类型参数的值限定为 Number 类或其子类
    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
    public static void main(String[] args) {
        Box3<Integer> integerBox = new Box3<Integer>();
        integerBox.add(new Integer(10));     // 正确：这里添加的是 Integer 对象
        integerBox.inspect("some text");     // 错误：这里还是 String!
    }
}
```

将泛型方法修改为包含限定类型参数，再编译上面的程序，编译将会失败，因为对inspect()方法的调用实际传递的是String类型，而不是泛型方法限定的Number类或其子类。下面是编译失败的信息：

```
Test.java:25: 无法将 Test<java.lang.Integer> 中的 <U>inspect(U) 应用于
(java.lang.String)
          integerBox.inspect("some text");          // 错误：这里还是String!
                     ^
1 错误
```

对类型参数的限定，也可以包括指定额外的必须被实现的接口，这使用&字符，如下所示，指定类型变量U可接收的值为实现了MyInterface接口的Number类或其子类。

```
<U extends Number & MyInterface>
```

在限定类型参数时，可以限定上限的类型中，最多只能有1个为类（也可以为0个），其余的必须均为接口。接口可以有多个，中间用&字符分隔。如下所示，指定类型变量U可接收的值为实现了接口MyInterface1、MyInterface2和MyInterface3的Number类或其子类。

```
<U extends Number & MyInterface1 & MyInterface2 & MyInterface3>
```

10.5 泛型子类型

在“继承”一章讲过，可以将一个类型的对象赋给另外一个类型的对象，只要两个类型是兼容的。例如，可以将一个Integer类型的赋给一个Object类型的，因为Object是Integer的一个父类型。

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;                    // OK
```

在面向对象的术语中，这被称为“是”的关系。因为Integer“是”Object的一种，所以这样的赋值是允许的。但是，Integer还是一种Number，因此下面的代码也是有效的。

```
public void someMethod(Number n){
    // 省略方法体
}
someMethod(new Integer(10));                 // OK
someMethod(new Double(10.1));                // OK
```

对于泛型，也可以这样使用。可以执行一个泛型调用，传递Number作为它的类型实参，而任何后来对于add方法的调用，如果参数与Number相兼容（是Number类型或其子类），都是可以的。

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));                    // OK
box.add(new Double(10.1));                   // OK
```

现在考虑下面的方法：

```
public void boxTest(Box<Number> n){
    // 方法体被省略
}
```

它接收哪种类型的参数呢？根据方法声明，它接收一个参数，参数的类型是Box<Number>。然而，这到底意味着什么呢？如期望的那样可以传递进 Box<Integer>或

Box<Double>吗？令人惊讶的是，答案是“不能”，因为 Box<Integer>和 Box<Double>不是 Box<Number>的子类型。如果想象一下切实的对象——可以实际地描述的事物——如笼子，理解为什么会这样就将变得很容易。例如：

```
// 一个笼子是一个事物的集合，带有格栅以防止事物掉出来.
interface Cage<E> extends Collection<E>;
```



Collection 接口是“集合类层次”的根接口，它代表一组对象。因为一个笼子可以被用于持有一个对象的集合（如动物），所以在本例中使用它更有意义。狮子是一种动物，所以 Lion 将是 Animal 的子类型。

```
interface Lion extends Animal{}
Lion king = …;
```

在需要动物的地方，可以引用一个狮子。例如：

```
Animal a = king;
```

一个狮子当然可以被放到一个关狮子的笼子里。例如：

```
Cage<Lion> lionCage = …;
lionCage.add(king);
```

而蝴蝶可以放到一个关蝴蝶的笼子里。例如：

```
interface Butterfly extends Animal {}
Butterfly monarch = …;
Cage<Butterfly> butterflyCage = …;
butterflyCage.add(monarch);
```

但是什么是一个“关动物的笼子”呢？或者准确地说，什么是一个“关所有动物的笼子”？如以下代码所示。

```
Cage<Animal> animalCage = …;
```

在这里，笼子被设计用来存放所有类型的动物，混合在一起。它必须有足够结实的格栅来关狮子，并且必须有足够小的空隙来关蝴蝶。这样的一个笼子甚至不可能真正地建造。但是如果可以的话，它既可以关狮子，也可以关蝴蝶。例如：

```
animalCage.add(king);
animalCage.add(monarch);
```

因为狮子是动物的一种（Lion 是 Animal 的子类型），所以问题就出现了，狮子的笼子是动物笼子的一种吗？Cage<Lion>是 Cage<Animal>的一个子类型吗？通过上面动物笼子的定义，答案肯定是“不”。这很令人惊讶！但是考虑下面这样的问题，将会有助于理解这个问题：一个狮子的笼子不能被假设可以装蝴蝶，同样地，一个蝴蝶的笼子也不能被假设可以用于装狮子。因此，两种笼子都不能被认为是一个“所有动物”的笼子。所以，下面的引用是错误的：

```
animalCage = lionCage;          //编译时错误
animalCage = butterflyCage;     //编译时错误
```

可以这样理解：Cage<Lion>和 Cage<Butterfly>不是 Cage<Animal>的子类型。

10.6　使用通配符

在泛型中，使用“通配符”字符“？”来代表一个未知的类型。例如，要指定一个能装“某些”类型动物的笼子，使用下面的代码形式。

```
Cage<? extends Animal> someCage = …;
```

将“? extends Animal”读做“一个未知的类型，它是 Animal 的子类型，也可能是 Animal 本身”，可以简称为“某些动物类型”。这是一个“限定通配符”的示例，在这里，Animal 形成所期望类型的“上限”。如果需要一个笼子，简单地装上“某些”类型的动物，那么可以自由地选择提供一个狮子的笼子“或者”一个蝴蝶的笼子。

需要注意的是，还可以通过使用 super 取代 extends 来指定一个“下限”。所以代码<? super Animal>可以将被读做“一个未知的类型，它是 Animal 的父类型，也可能是 Animal 本身”。还可以使用一个“无限定通配符”<?>来指定一个未知类型。无限定通配符本质上与<? extends Object>相同。

Cage<Lion>和 Cage<Butterfly>不是 Cage<Animal>的子类型，实际上它们是 Cage<? extends Animal>的子类型。例如：

```
someCage = lionCage;            //OK
someCage = butterflyCage;       //OK
```

所以，现在问题变为“能直接将蝴蝶和狮子添加到 someCage 中吗？”。可以猜得出来，答案是“不”。

```
someCage.add(king);             //编译时错误
someCage.add(monarch);          //编译时错误
```

如果 someCage 是一个蝴蝶的笼子，那么它存放蝴蝶很合适，但是如果放狮子的话，狮子将会破窗而出。如果 someCage 是一个狮子的笼子，那么用它来装狮子将会很合适，但是如果用来装蝴蝶的话，蝴蝶就会飞走。这样看来，根本不能放任何东西到 someCage 中，它是不是没有用处呢？并不是没有用处，因为仍然可以读出它的内容。例如：

```
void feedAnimals(Cage<? extends Animal> someCage>){
    for(Animal a : someCage)
        a.feedMe();
}
```

所以，可以像刚开始讲的那样，将各种动物放到它们单独的笼子里，并首先为狮子调用这个方法，再为蝴蝶调用这个方法。例如：

```
feedAnimals(linoCage);
feedAnimals(butterflyCage);
```

或者，选择将所有的动物组合到所有动物的笼子里。例如：

```
feedAnimals(animalCage);
```

10.7 类型擦除

当一个泛型被实例化时，编译器通过称为“类型擦除”的技术来编译这些类型。所谓类型擦除，指的是编译器移除一个类或方法中所有与类型参数相关的信息。类型擦除能够使使用泛型的 Java 应用程序与 Java 类库和在泛型出现之前创建的应用程序保持二进制上的兼容性。

例如，Box<String>被编译为类型 Box，在这里，Box 被称为“原类型（raw type）”。所谓原类型，指的是不带任何类型参数的泛型类或接口的名称。这意味着无法找出一个泛型类在运行时正在使用的是 Object 的什么类型。像下面这样的操作是不可能的：

```
public class MyClass<E> {
    public static void myMethod(Object item) {
        if (item instanceof E) {          //编译器错误
            ...
        }
        E item2 = new E();                //编译器错误
        E[] iArray = new E[10];           //编译器错误
        E obj = (E)new Object();          //未经检查的类型转换警告
    }
}
```

在上面的代码中，所有含有 E 的语句在运行时是无意义的，因为编译器在编译时移除所有与实际类型参数相关的信息（这里是类型参数 E）。

因为存在着类型擦除，所以新的代码可能继续使用遗留的代码。但是不管什么原因，使用一个原类型被认为是一个不好的程序设计实践，只要有可能，就应该尽量避免使用。当混合使用遗留代码与泛型代码时，可能会遇到类似下面这样的警告信息：

```
注意：警告 Demo.java 使用了未经检查的或不安全的操作.
注意：使用-Xlint:unchecked 重新编译以获得更详细的信息.
```

当使用一个操作原类型的过时的 API 时，常常会发生这种情况，如下面的 WarningDemo 程序所显示的一样：

```
public class WarningDemo {
    public static void main(String[] args){
        Box<Integer> bi;              //声明 Box 实例变量，接收 Integer 类型参数
        bi = createBox();             //调用 createBox 方法，创建一个 Box 类型对象
    }
    // 假设这个方法是过时的类库的一部分，在泛型出现之前编写的。它返回 Box 而不是 Box<T>
    static Box createBox(){
        return new Box();
    }
}
```

使用-Xlint:unchecked 重新编译，会出现下面这样的额外的信息：

```
警告 Demo.java:4: warning: [unchecked] unchecked conversion
找到：Box
要求：Box<java.lang.Integer>
        bi = createBox();
                    ^
1 警告
```

在这个程序中，类 WarningDemo 中混合使用了泛型代码和原类型代码。程序期待的是 Box<java.lang.Integer>类型，而对方法 createBox()的调用返回的却是 Box，从而出现警告信息。

10.8　小结

本章主要描述了什么是泛型、为什么要使用泛型及如何使用泛型。读者在学习本章时，重点理解什么是泛型及为什么要使用泛型。读者可能在学习本章的过程中对如何使用泛型还存在迷惑，没有关系，完全可以先不去理会它。在具体使用泛型最多的地方一般是在 Java 的集合框架部分，所以读者可以在学习 Java 集合框架部分内容时，再回来结合本章的内容来掌握泛型的使用。下一章向读者介绍包的概念和使用。

第11章 包

“包”指的是一组提供访问保护和命名空间管理的相关的类型。注意这里的“类型（types）”指的是类、接口、枚举及注解类型。枚举和注解类型分别是特殊的类和接口，所以在本章中，涉及“类型”时简单地指类和接口。本章讲解如何将类和接口封装到一个包中、如何使用包中的类，以及如何组织自己的文件系统使得编译器能够找到源文件。



11.1 使用包

试想一下：如果一个具有100多人的单位，没有任何的部门和科室，管理一定会陷于混乱。同样，当一个程序中包含100多个类时，如果不将它们按其用途分门别类组织在不同的文件夹中，那么开发工作也将陷于一片混乱。为了使得各种类型易于查找和使用，避免命名冲突，并控制访问，程序员将一组相关的类型封装到包（package）中。

当计算机中的文件越来越多时，一定不会将它们全部放置在同一目录下，而会按其用途的不同放置在不同的文件夹中。与此相似，作为Java平台一部分的各种类型，也都是按功能封装类的各种包的成员：基本类在java.lang包中，用于读和写（输入和输出）的类在java.io包中，等等。程序员也可以将自己编写的各种类型放到包中。

假设程序员自己编写了一组代表图形对象的类，如圆、矩形、直线和点。另外还编写了一个接口Draggable，可以使用鼠标拖动的类要实现这个接口。如下所示。

```
//在Draggable.java文件中
public interface Draggable {
    . . .
}
//在Graphic.java文件中
public abstract class Graphic {
    . . .
}
//在Circle.java文件中
public class Circle extends Graphic implements Draggable {
    . . .
}
//在Rectangle.java文件中
```

```
public class Rectangle extends Graphic implements Draggable {
    . . .
}
//在 Point.java 文件中
public class Point extends Graphic implements Draggable {
    . . .
}
//在 Line.java 文件中
public class Line extends Graphic implements Draggable {
    . . .
}
```

在应用过程中，应该将这些类和接口封装到一个包中，从而更好地管理已经开发的 Java 代码。这样有以下好处：

- 程序员能很容易地判断出这些类型是相关的。
- 程序员知道到哪里能找到提供图形相关功能的类型。
- 程序员自己创建的类型的名称将不会和其他包中的类型的名称相冲突，因为包创建了一个新的命名空间。
- 可以允许在此包中的类型彼此自由地相互访问，同时仍能限制对包外的类型的访问。



11.2　创建包

要创建一个包，需要为包选择一个名称，并将一个 package 语句和包的名称放在每一个想要放到包中的源文件（包含各种类型：类、接口、枚举和注解类型）的顶部。其中 package 为关键字。所有的包名都是小写的。

11.2.1　创建包的语句

使用含有关键字 package 的语句来创建包。例如，在上一节创建的各种与图形相关的类，都放到同一个包 graphics 中，那么需要在每一个源文件中的顶部包含以下代码：

```
package graphics;
```

在这里，package 语句必须是源文件中的第一行。在每一个源文件中只能有一个 package 语句，它适用于文件中的所有类型。

如果在一个源文件中有多个类型，那么只有一个允许是 public 的，并且它必须与源文件同名。例如，可以在 Circle.java 文件中定义 public class Circle 类，在 Draggable.java 文件中定义 public interface Draggable 接口，在 Day.java 文件中定义 public enum Day 枚举，等等。

在含有 public 类型的文件中也可以包含非公共类型（强烈建议不要这样做，除非此非公共类型很小并且与 public 类型密切相关），但只有 public 类型可以从包外被访问到。所有顶层的、非公共类型都将是“包级私有（package-private）”的。

如果将上一节列出的与图形相关的接口和类放到一个名为“graphics”的包中，需要 6 个源文件，如下所示。

```
//在 Draggable.java 文件中
package graphics;
public interface Draggable {
```

```
    . . .
}
//在 Graphic.java 文件中
package graphics;
public abstract class Graphic {
    . . .
}
//在 Circle.java 文件中
package graphics;
public class Circle extends Graphic implements Draggable {
    . . .
}
//在 Rectangle.java 文件中
package graphics;
public class Rectangle extends Graphic implements Draggable {
    . . .
}
//在 Point.java 文件中
package graphics;
public class Point extends Graphic implements Draggable {
    . . .
}
//在 Line.java 文件中
package graphics;
public class Line extends Graphic implements Draggable {
    . . .
}
```

如果不使用 package 语句，那么类型都属于一个未命名的包。一般而言，未命名的包只用于小的或临时的应用程序，或者刚刚开始开发过程时。除此之外，类和接口都应放到命名的包中。

11.2.2 命名包

在全世界同时会有很多的程序员都使用 Java 程序设计语言来编写类和接口，所以很有可能许多程序员将使用相同的名称来命名不同的类型。实际上，在上一节的例子就是这样的：作者定义了一个 Rectangle 类，同时在 java.awt 包中已经有了一个 Rectangle 类。

尽管如此，Java 编译器允许在不同的包中的两个类可以有相同的名称，但它们的全限定名称是不同的。每一个 Rectangle 类的全限定名称包括包名，也就是说，在 graphics 包中的 Rectangle 类的全限定名称是 graphics.Rectangle，而在 java.awt 包中的 Rectangle 类的全限定名称是 java.awt.Rectangle。

这样命名包的方式通常没有问题，除非两个独立的程序员分别为他们的包使用了相同的名称。要防止出现这样的问题，在命名包时要遵循包命名惯例。

11.2.3 包命名惯例

包名使用小写字母，以避免与类名或接口名冲突。对公司来说，使用其颠倒的 Internet 域名来开始其包名。例如，example.com 的程序员创建的 images 包要命名为 com.example.images。

在一个公司内部发生的命名冲突，需要按照该公司的规定来处理。例如通过在公司名后包含地区或项目名称。例如，命名为 com.company.region.package。Java 语言本身的包以 java 或 javax 开始。

在有的情况下，互联网的域名可能不是一个有效的包名。例如，如果域名包含一个连字符或其他特殊字符，如果包名以数字开头或者以其他用于 Java 名称开头时是非法的字符开头，或者如果包名包含一个保留的 Java 关键字，如“int”，那么这种情况就会发生。在这种情况下，建议添加下画线。如表 11.1 所示。

表 11.1　合法的包名

域名	包名前缀
clipart-open.org	org.clipart_open
free.fonts.int	int_.fonts.free
poetry.7days.com	com._7days.poetry

11.3　使用包中的成员

组成包的类型称为“包的成员”，包括类、接口、枚举和注解类型，都是包的组成成员。从一个包的外部使用一个 public 包成员，必须满足以下条件之一：

- 通过全限定名指向包的成员。
- 导入包成员。导入成员所在的整个包。

11.3.1　通过全限定名引用包成员

迄今为止，本书所用到的示例基本上都是通过简单的名称引用类型，如 Rectangle 和 AutoCars。如果所编写的代码与要引用的成员在同一个包中，或者要引用的成员已经导入，可以使用包成员的简单名称。

但是，如果说想使用另外不同的包中的成员，并且该包没有被导入，就必须使用成员的全限定名，包括包名。例如，要使用上一节示例中 graphics 包中声明的 Rectangle 类，需要使用下面所示的全限定名：

```
graphics.Rectangle
```

可以使用这个全限定名创建一个 graphics.Rectangle 类的实例：

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

全限定名是很少使用的。当经常使用一个名称时，重复地输入限定名是很乏味的，而且代码变得难以阅读。这时，可以“导入”包或包中的成员，然后在代码中只使用简单名称即可。

11.3.2　导入包中的成员

要将一个指定的成员导入到当前文件中，需要在文件的开始放置一个 import 语句。import 语句位于 package 语句之后，任何类型的定义之前。例如，要导入前面所讲的 graphics 包中的 Rectangle 类，使用下面的语句：

```
import graphics.Rectangle;
```

导入以后，在代码中可以使用简单名称来引用 Rectangle 类：

```
Rectangle myRectangle = new Rectangle();
```

这适合于从 graphics 包中导入少量的成员。如果使用到该包中许多类型的话，应该导入整个包。

11.3.3 导入整个包

有时候，在编写程序时，需要用到某些包中很多的类型，这时可以不用一个类型一个类型地导入，而要使用带有通配字符“*”（星号）的 import 语句导入一个特定包中包含的所有类型，如下面的代码所示。

```
import graphics.*;
```

通配符*代表该该包中所有的类型，但不包括包含的子包。导入以后，就可以通过简单名引用 graphics 包中的任何类或接口，如下面的代码所示。

```
Circle myCircle = new Circle();          //使用 graphics 包中的 Circle 类
Rectangle myRectangle = new Rectangle(); //使用 graphics 包中的 Rectangle 类
```

在这里，import 语句中的星号（*）只被用于指定包中的所有类。它不能被用于匹配一个包中的类的子集。例如，下面的代码中，星号（*）并不匹配 graphics 包中以 A 开头的类。

```
import graphics.A*;                              //错误的用法
```

这时会引起编译错误。使用 import 语句，一般只能导入一个具体的包的成员，或者整个包。另外，有一种较少使用的普通的 import 形式，用来导入一个封装类的 public 形式的嵌套类。例如，如果 graphics.Rectangle 类包含有用的嵌套类，如 Rectangle.DoubleWide 和 Rectangle.Square，那么可以使用以下两个语句导入 Rectangle 和它的嵌套类：

```
import graphics.Rectangle;
import graphics.Rectangle.* ;
```

其中第二个导入语句将不会导入 Rectangle 类。还有另外一种较少使用的普通的 import 形式，“静态导入语句”，在本节结束时讲解。

为了方便，Java 编译器自动为每个源文件导入三个完整的包：

- 没有名称的包。
- java.lang 包。
- 当前包（当前文件所在的包）。

11.3.4 包的层级

表面看来包是有层级的，但是实际上它们没有这样的层级关系。例如，Java API 包含有一个 java.awt 包，一个 java.awt.color 包，一个 java.awt.font 包，以及许多其他的以 java.awt 开头的包。但是，java.awt.color 包、java.awt.font 包和其他 java.awt.xxxx 包并不包含在 java.awt 包中。前缀 java.awt（Java 抽象窗口工具箱）被用于许多与桌面程序设计相关的包，以使其关系更加明显，但是并不表示它们有内含的关系。

导入 java.awt.*，会导入 java.awt 包中的所有类型，但是不会导入 java.awt.color 包、java.awt.font 包或者任何其他的 java.awt.xxxx 包。如果既需要使用 java.awt.color 包中的类和其



他类型，又需要 java.awt 包中的类和其他类型，那么就必须导入两个包中的所有文件。例如：

```
import java.awt.* ;
import java.awt.color.* ;
```

11.3.5　包成员名称冲突

如果一个包中的一个成员与另外一个包中的一个成员具有相同的名称，那么必须通过全限定名使用每一个成员。例如，在 graphics 包中定义一个名为 Rectangle 的类。在 java.awt 包中也包含一个 Rectangle 类。如果 graphics 和 java.awt 包都被导入到同一个文件中，那么像下面这样的代码就是不确定的：

```
Rectangle rect ;
```

在这种情况下，必须使用成员的全限定名称来确切指出想使用哪个 Rectangle 类。例如：

```
graphics.Rectangle rect ;
```

11.3.6　静态导入语句

有时需要频繁地访问某个（或某些）类中的 static final 字段（常量）和静态方法。反复地将这些类的名称作为前缀会导致代码混乱。“静态导入”语句提供了导入常量和静态方法的一种方式，这样就不需要使用它们的类名作为前缀。

例如，在 Java API 中有个 java.lang.Math 类，它定义了常量 PI（代表圆周率）及许多静态方法，包括用来计算正弦、余弦、正切、平方根、最大值、最小值、指数等的方法。例如：

```
public static final double PI 3.141592653589793
public static double cos(double a)
```

通常，如果在另外一个类中使用当前类中声明的这些对象，需要使用类名作为前缀。例如，在 Math 类中定义了很多与数学运算有关的静态方法，如果要调用的话，如下面的代码所示。

```
double r = Math.cos(Math.PI * theta);  //调用 Math 类的静态方法求余弦及圆周率
```

如果频繁地使用 Math 类的静态方法和静态字段 PI，每次都使用 Math 前缀很烦琐，这时就可以使用静态导入语句来导入 java.lang.Math 中的静态成员，然后就不需要再在代码中使用类名 Math 作为前缀。Math 类的静态成员既可以单独地被导入，例如：

```
import static java.lang.Math.PI;
```

也可以成组地被导入，例如：

```
import static java.lang.Math.*;
```

一旦静态成员被导入，就可以不带限定名地使用它们。例如，前面求余弦的代码片段可以像下面这样来写，而不必再通过类名 Math 来调用。

```
double r = cos(PI * theta);
```

很显然，程序员可以编写自己经常使用的包含有常量和静态方法的类，然后在程序中使用静态导入语句，实际上，这种方式在开发中经常用到。例如：

```
import static MyConstants.*;
```

需要注意的是，要慎用静态导入。过度地使用静态导入将会使代码难以阅读和维护，

因为代码的读者并不知道哪个类定义了特别的静态对象。适当地使用静态导入，通过消除类名的重复，将会使代码可读性更强。



11.4 管理源文件和 class 类文件

Java 平台的许多实现依赖于分组的文件系统来管理源文件和 class 文件（虽然 Java 语言规范对此没有要求）。例如，包名对应着系统中实际的目录名。包括在安装 JDK 时，所生成的目录也是与类所属的包路径相对应的。通过一定的策略来管理源文件和 class 文件，会使开发更加有序和高效。本节介绍管理源文件和 class 类文件的实现策略。

11.4.1 管理实现策略

将一个类、接口、枚举或注解类型的源代码放到一个文本文件中，文本文件的名称是类型的简单名称，扩展名为.java。例如：

```
// 在 Rectangle.java 文件中
package graphics;
public class Rectangle() {
   . . .
}
```

然后，将该源文件放到与该类型所属的包名相对应的目录中，例如：

```
.....\graphics\Rectangle.java
```

包成员的限定名和文件的路径名是平行的。在微软 Windows 中，文件名分隔符为反斜线（对于 UNIX 系统，使用正斜线）。

```
类名：              graphics.Rectangle
文件路径名：        graphics\Rectangle.java
```

前面讲过，按惯例，一个公司使用其反转的 Internet 域名用做包名。例如 Example 公司，它的 Internet 域名是 example.com，就会在其所有的包名前使用 com.example。包名的每一个组成部分对应一个子目录。因此，如果 Example 公司有一个 com.example.graphics 包，包含有一个 Rectangle.java 源文件，那么它就应包含一系列的子目录，像下面这样：

```
....\com\example\graphics\Rectangle.java
```

当编译源文件时，编译器为文件中所定义每一个类型都创建一个不同的输出文件，输出文件的基名是类型的名称，扩展名为.class。例如，如果有下面这样的源文件：

```
// 在 Rectangle.java 文件中
package com.example.graphics;
public class Rectangle{
      . . .
}
class Helper{
      . . .
}
```

那么编译过的文件将位于以下位置：

```
<输出文件的父目录路径>\com\example\graphics\Rectangle.class
<输出文件的父目录路径>\com\example\graphics\Helper.class
```



像.java 源文件一样，编译过的.class 文件也应该位于与包名相对应的一系列的目录下。不过，.class 文件的路径不一定非要和.java 源文件的路径一样。例如，可以分别组织自己的源文件和 class 文件目录，如下所示。

```
<path_one>\sources\com\example\graphics\Rectangle.java
<path_two>\classes\com\example\graphics\Rectangle.class
```

这样组织的话，程序员就可以将 classes 目录给其他的程序员使用，而不必暴露自己的源代码。此时程序员还需要照这样来管理源文件和 class 文件，然后编译器和 Java 虚拟机（JVM）能够找到程序中使用的所有类型。

classes 目录的全路径，<path_two>\classes，被称为“类路径(classpath)”，使用 CLASSPATH 系统变量来设置类路径。编译器和 JVM 都通过添加包名到类路径来构造.class 文件的路径。例如，如果用户当前的类路径是：

```
<path_two>\classes
```

同时包名是：

```
com.example.graphics
```

那么编译器和 JVM 会在下面的路径中查找.class 文件：

```
<path_two>\classes\com\example\graphics
```

类路径可能包括好几个路径，由分号（Windows 系统）分隔或冒号（UNIX 系统）分隔。默认情况下，编译器和 JVM（Java 虚拟机）搜寻当前目录及包含 Java 平台类的 JAR 文件，因此这些目录会自动地出现在类路径中。

11.4.2　设置 CLASSPATH 系统变量

要显示当前 CLASSPATH 变量，在 Windows 和 Unix 中使用下面这样的命令：

```
Windows 中:   C:\> set CLASSPATH
Unix 中:      % echo $CLASSPATH
```

要删除 CLASSPATH 变量的当前内容，使用下面的命令：

```
Windows 中:   C:\> set CLASSPATH=
Unix 中:      % unset CLASSPATH;export CLASSPATH
```

要设置 CLASSPATH 变量，使用下面的命令（示例，根据用户的实际情况设定）：

```
Windows 中:   C:\> set CLASSPATH=C:\users\george\java\classes
Unix 中:      % CLASSPATH=/home/george/java/classes;export CLASSPATH
```

11.5　小结

本章主要介绍了 Java 语言中包的概念及如何创建和使用包。包是 Java 组织资源的一种方式，读者学习起来应该不会感到吃力。只是如何将包对应到具体的物理路径上，读者应该细心体会。因为 Java 平台的许多实现依赖于分组的文件系统来管理源文件和 class 文件的，包名往往对应着系统中实际的目录名，这一点读者一定要注意。下一章将介绍 Java 语言中非常重要也经常使用的集合框架。

第12章 集合

Java 集合框架是 Java 语言的重要组成部分，包含有系统而完整的集合层次体系，封装了大量的数据结构的实现。深刻理解 Java 集合框架的组成结构及其中的实现类和算法，会极大提高程序员编码的能力。本章讲述 Java 集合框架，主要包括集合的概念、使用集合的优点，以及组成 Java 集合框架的核心元素：接口、实现和算法。



12.1 Java 集合框架概述

集合有时又称为容器，简单地说，它是一个对象，能将具有相同性质的多个元素汇聚成一个整体。集合被用于存储、获取、操纵和传输聚合的数据。典型地，集合代表形成一个自然组合的数据条目，如一副纸牌（一个纸牌卡片的集合）、一个邮包（一个信函的集合）或者一个电话本（一个姓名和电话号码的映射集合）。

在 Java 1.2 以前的版本中，集合的实现包括 Vector（向量）、Hashtable（哈希表）和 Array（数组）。不过这些早期的版本仅实现了一些集合类，并没有包含一个完整的集合框架。

12.1.1 Java 集合框架

集合框架（Collections Framework）是用来表现和操纵集合的一个统一的体系结构。所有的集合框架都包含以下内容。

- 接口：是代表集合的抽象数据类型。接口允许独立于其表现细节地操纵集合。在面向对象的语言中，接口通常形成了一个有继承关系的层级结构。
- 实现：是集合接口的具体实现。本质上，它们是可重用的数据结构。
- 算法：是在实现了集合接口的对象上执行有用的计算的方法，如查找和排序。算法通常是“多态的”，就是说，同样的方法可以在同一个接口被多个类实现时有不同的表现。本质上，算法是可重用的函数。

除了 Java 集合框架之外，最有名的集合框架的例子是 C++的标准模板库（STL）和 Smalltalk 的集合层级。从历史上看，集合框架已经相当复杂，因此普遍认为，学习集合框架有一个很陡峭的学习曲线。但是 Java 的集合框架将打破这种传统，如读者将在本章学习到的一样，它将很容易被掌握。

12.1.2　使用 Java 集合框架的益处

对程序员来说，使用 Java 集合框架具有以下的优点。

- 降低编程的难度：通过提供有用的数据结构和算法，集合框架使得程序员将精力集中在程序的重要部分，而不是为了让程序能正常运转而将注意力放在低层设计上。Java 集合框架将程序员从编写适配器对象等大量的代码编写这种情况下，转移到如何组合使用集合 API 上。
- 提高程序编写的速度和质量：对于经常使用的数据结构和算法，Java 集合框架提供了高性能、高质量的实现以提高程序的开发速度和质量。因为程序员被从编写自己的数据结构这样的苦差使上解脱出来，所以他们有更多的时间用于提高程序的质量和性能上。
- 允许在不相关联的 API 中协同工作的能力：集合接口是 API 间来回传递集合的专用语言。如果一个网络管理 API 提供一个网络结点名称的集合，而一个 GUI 工具想要的是一个列标题的集合，那么即使这些 API 是各自独立编写的，它们也应该无缝地协同工作。
- 降低学习和使用新 API 的难度：很多 API 天生的有对集合的存储和获取的功能。在过去，每个 API 都有一些小的子 API，专门用于操纵各自的集合。在这些特殊的集合子 API 之间就会缺乏一致性，因此程序员不得不匆匆学习每一种，并且在使用时很容易出错。而随着标准集合接口的出现，这些问题迎刃而解。
- 降低设计新 API 的难度：相对于前面的优点，这个优点似乎不是很重要。当设计者和实现者想要创建一个依赖于集合的新 API 时，不需要再重新设计，只需要使用标准集合接口即可。
- 鼓励软件重用：符合标准集合接口的新的数据结构天生具有可重用性。同样，对于操作对象（实现了这些集合接口）的算法也是如此。

12.2　核心集合接口

“核心集合接口（core collection interfaces）”是 Java 集合框架的灵魂。“核心集合接口”封装了不同类型的集合，如图 12-1 所示。

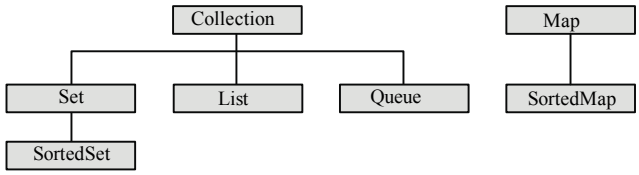


图 12.1　核心集合接口

可以独立于这些接口的表现细节来操纵它们。核心集合接口是 Java 集合框架的基础。正如在图 12.1 中看到的，这些核心集合接口形成了层级关系。

其中，Set 是 Collection 的子接口，所以它是一种特殊的 Collection，同样 SortedSet 是一种特殊的 Set，等等。要注意的是，虽然这个集合接口层级由两个不同的树组成，但实际上 Map 并不是真正的 Collection。另外要注意的是，所有的核心集合接口都是泛型的。

例如，下面是 Collection 接口的声明：

```
public interface Collection<E>...
```

由语法<E>可知，这个接口是个泛型。当声明一个 Collection 的实例时，可以指定并且应该指定要放入集合中的对象的类型。指定类型可以使编译器在编译时检查放入集合中的对象的类型是否正确，从而减少运行时错误。

一旦理解了怎样使用这些接口，也就掌握了 Java 集合框架的大部分。本章主要讲解有效地使用这些接口的一般指导原则，包括什么时候用哪一个接口。还会讲到针对每一个接口的编程习惯用法，帮助读者快速掌握其基本用法。

为了保持核心集合接口的数量，以利于管理，Java 平台并不为每一个集合类型的每一变种提供另外的接口（各种变种可能包括不可变的集合、固定大小的集合及只能读取的集合等）。Java 采用了另外一种方式，在具体实现每个接口时，将其具体实现设计为带有选择性的，即选择支持某些操作而限制另外一些操作。

例如，某个具体实现可能会选择支持元素顺序排列，同时限制对元素可重复性的检查操作。或者另外的某些具体实现可能会选择既支持对元素排序，又限制元素重复。因此一个给定的实现可能并不选择支持所有的操作。如果调用了一个不被支持的操作，那么集合会抛出 UnsupportedOperationException 类型的异常。

在 Java 平台中，所有通用的实现支持所有可选的操作。下面描述了核心集合接口中的各个接口类型：

- Collection——集合接口层级的根。一个集合代表一组对象的聚合，这些对象称为这个集合的“元素”。Collection 接口提供了一些最通用的方法，通过这些方法可以实现对集合的基本操作。有的集合允许重复的元素，而其他的集合不允许有重复的元素。有些集合是有序的，有些集合是无序的。Java 平台并不提供任何对 Collection 接口的直接实现，但是提供更加详细的对子接口的实现，诸如 Set 和 List 接口。
- Set——不能包含重复元素的集合。这个接口是数学集合的抽象模型，被用于代表集合，如组成一副牌的扑克牌，组成一个学生课程表的课程，或者运行在一个机器上的处理器等。
- List——一个有序的集合。List 可以包含重复的元素，并且 List 中的元素是有序排列的。因此使用 List 通常需要对列表中每个元素的插入位置进行精确的控制，并且能通过元素的整数索引（位置）访问元素。
- 队列——是一个其元素带有先后处理顺序的集合。除了基本的 Collection 操作，队列还提供额外的插入、取出和检查操作。队列通常（但不是必需的）以一种 FIFO（先进先出）的形式管理元素。但优先级队列例外，它根据所提供的比较方法或元素的自然顺序来对元素进行排序。不论使用哪种顺序，排在队列的第一个的元素是在调用 remove 或 poll 方法时要移除的元素。在一个 FIFO 队列中，所有新的元素被插入到队列的尾部。其他类型的队列可能使用不同的放置规则。每一个 Queue 的实现必须指定它的排序属性。
- Map——一个将键映射到值的对象。Map 不能包含重复的键；每个键最多能映射到一个值。

最后，还有两个核心集合接口，它们是 Set 和 Map 的子接口，是有序的 Set 和 Map：

- SortedSet——元素按升序排列的 Set。还增加了很多额外的操作来利用其排序的特



性。有序的集合用于自然顺序的集合，如单词列表和会员目录等。

- SortedMap——其中的映射关系按键的升序排列的 Map。这与 SortedSet 相似。有序的映射用于自然顺序的键值对集合，如字典和电话号码簿。



12.3　Collection 接口

一个集合代表一组对象，这些对象作为集合的元素。在 Java 集合框架中定义的 Collection 接口，提供了集合接口的通用操作，通常用于传递集合对象。Collection 接口是 Sort 接口和 List 接口的父接口，提供了大量的通用的集合操纵方法。

12.3.1　转换构造器

按惯例，所有的 Collection 接口的实现都有一个带有一个集合参数的构造器。这个构造器，作为“转换构造器”，将新的集合初始化为包含集合参数的所有元素，不论是给定接口的子接口或其实现类型。换句话说，这允许程序员来“转换”集合的类型。例如，假设有这么一个集合：

```
Collection<String> c;
```

这里的 c 可以是一个 List，一个 Set，或者另外一种 Collection。通常习惯地创建一个新的 ArrayList（List 接口的一个实现），初始化为包含 c 中的所有元素。例如：

```
List<String> list = new ArrayList<String>(c);
```

这里的 ArrayList 是 List 接口的一个实现类，在创建其实例对象时，将一个集合 c 作为参数传递给其构造器，有 c 中的所有元素初始化 list。也就是说，通过使用“转换构造器”，list 对象就包含了集合 c 中的所有元素。

12.3.2　Collection 接口的定义

下面是 Collection 接口的定义。通过了解接口的定义，掌握在 Collection 接口中定义的通用的方法，这些通用的方法同样会被 Collection 接口的子接口所继承。

```
public interface Collection<E> extends Iterable<E> {
    // 基本操作
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);                          //可选的
    boolean remove(Object element);                  //可选的
    Iterator<E> iterator();
    // 批量操作
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);  //可选的
    boolean removeAll(Collection<?> c);         //可选的
    boolean retainAll(Collection<?> c);         //可选的
    void clear();                                //可选的
    // 数组操作
    Object[] toArray();
    <T> T[] toArray(T[] a);
```

```
}
```

在 Collection 接口中，定义了对于集合元素进行基本操作、批量操作和数组操作的方法。

12.3.3 Collection 接口的基本操作

从 Collection 的定义可以看出，Collection 接口的基本操作有 6 个。使用这些方法，可以知道集合中有多少个元素（即集合的大小，size()和 isEmpty()方法），可以检查集合中是否包含某个元素（contains()方法），可以向集合中添加元素和从集合中移除元素（add()方法、remove()方法），以及获取访问集合元素的一个迭代器（iterator()方法）。

其中，add 方法被定义得相当通用，所以它对于允许重复元素的集合和不允许重复元素的集合都一样有意义。该方法保证在调用结束以后，Collection 将包含指定的元素，并且如果调用的结果是 Collection 发生了改变，那么返回 true。同样地，remove()方法从 Collection 移除指定的元素，并且如果作为该方法调用的结果 Collection 被修改了，那么返回 true。

12.3.4 遍历 Collection 接口

要遍历集合中的元素，有两种方法：使用 for-each 结构和通过使用 Iterator 迭代器。下面依次讲解这两种方法。

1．for-each 结构

for-each 结构允许使用 for 循环简洁地遍历一个集合或数组。下面的代码使用 for-each 结构在每一行上输出一个集合的每一个元素：

```
for (Object o : collection)
    System.out.println(o);
```

2．Iterator 迭代器

迭代器是一个对象，通过它可以遍历一个集合并从集合中有选择地移除元素。通过调用集合的 iterator 方法来获得集合的迭代器。接口 Iterator 的定义如下。

```
public interface Iterator<E> {
    boolean hasNext();          //判断是否有下一个元素
    E next();                   //获得下一个元素
    void remove();              //删除当前的元素。这个方法是可选的
}
```

当迭代器中还有更多的元素时，hasNext 方法返回 true，next 方法返回迭代器中的下一个元素。remove 方法移除由 next 方法从 Collection 中返回的最后一个元素。每次调用 next 方法以后只能调用 remove 方法一次，如果违犯此规则，就会抛出一个异常。

需要注意的是，在迭代期间，Iterator.remove 是修改集合的唯一安全的方法；在迭代期间，任何其他修改集合的方式都会造成不确定的结果。在下面几种情况下，使用 Iterator 代替使用 for-each 结构：

- 移除当前元素。for-each 结构隐藏迭代器，因此不能调用 remove 方法。所以，for-each 结构不能用于清除集合元素。
- 在多重集合上进行并行迭代。

下面的方法演示了怎样使用一个 Iterator 来过滤一个任意的 Collection 集合，即遍历集



合并移除指定的元素。

```
static void filter(Collection<?> c) {
   for (Iterator<?> it = c.iterator(); it.hasNext(); )
      if (!cond(it.next()))
         it.remove();
}
```

其中，调用集合 c 的 iterator 方法，获得一个迭代器 it；通过迭代器 it 的 hasNext 方法来判断是否还有更多的元素。如果有，返回 true，循环继续。在循环体内，通过调用迭代器 it 的 next 方法，来获取迭代器中的下一个元素。

这个简单的代码片段是多态的，这意味着它可以用于过滤任何类型的集合，而不需考虑具体实现。从这个例子中可以看出，使用 Java 集合框架来编写一个多态的算法很容易。

12.3.5　Collection 接口的批量操作

批量操作指在整个集合上的操作。可以通过使用基本操作来实现这些快速操作，不过在大多数情况下，这样的实现是缺乏效率的。下面是批量操作的一些方法：

- containsAll——如果目标集合包含指定集合的所有元素，返回 true。
- addAll——将指定集合中的所有元素添加到目标集合中。
- removeAll——从目标集合中移除所有同时还包含在指定集合中的元素。
- retainAll——从目标集合中移除所有没有包含在指定集合中的元素。也就是说，它只在目标集合中保留在指定集合中含有的元素。
- clear——从集合中移除所有元素。

如果在执行相应操作的过程中，目标集合被修改了，那么 addAll、removeAll 和 retainAll 方法都返回 true。下面是一个表现批量操作强大功能的示例，从一个名为 c 的 Collection 中移除一个指定元素 e 的所有实例。

```
c.removeAll(Collections.singleton(e));
```

在更明确的情况下，例如想要从一个 Collection 中移除所有的 null 元素，可以使用下面的语句：

```
c.removeAll(Collections.singleton(null));
```

Collections.singleton 是一个静态工厂方法，返回一个只包含指定元素的不可变的 Set 集合。例如 Collections.Singleton(e)方法返回只包含元素 e 的 Set 集合，然后集合 c 调用 removeAll 方法删除 c 中所有元素 e 的实例。同样，Collections.Singleton(null)方法返回只包含元素 null 的 Set 集合，然后集合 c 调用 removeAll 方法删除 c 中所有 null 元素。

12.3.6　Collection 接口的数组操作

toArray()方法主要是作为集合和老的期望输入数组的 API 之间的桥梁。数组操作允许 Collection 中的内容被转换到一个数组中去。这种不带参数的简单的方式创建一个新的 Object 数组。例如，假设 c 是一个 Collection，下面的代码片段将 c 中的内容放入一个新分配的 Object 数组当中，该数组的长度等于 c 中元素的数量。

```
Object[ ]  a = c.toArray( );
```

有一种更复杂的形式，允许调用者提供一个数组或选择输出数组的运行时类型。例如，假设 c 已知只包含字符串。下面的代码片段将 c 中的内容放入一个新分配的 String 数组当中，该数组的长度等于 c 中元素的数量。

```
String[ ]  a = c.toArray(new String[0]);
```



12.4 Set 接口

Set 是一个不能包含重复元素的接口。它是数学集合的抽象模型。Set 接口是 Collection 的子接口，只包含从 Collection 继承过来的方法，并增加了对 add()方法使用的限制，不允许有重复的元素。Set 还修改了 equals()和 hashCode()方法的实现，允许对 Set 实例进行内容上的比较，即便它们的实现类型不同。如果两个 Set 实例包含相同的元素，那么它们就是相等的。

12.4.1 Set 接口的定义

下面是 Set 接口的定义。凡是实现了 Set 接口的类，都要实现 Set 接口中的方法。因此先了解一下 Set 接口的定义，有助于快速并且全面地掌握这些方法的用法。Set 接口的定义如下所示。

```
public interface Set<E> extends Collection<E> {
    // 基本操作
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);                          //可选的
    boolean remove(Object element);                  //可选的
    Iterator<E> iterator();
    // 批量操作
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);  //可选的
    boolean removeAll(Collection<?> c);          //可选的
    boolean retainAll(Collection<?> c);          //可选的
    void clear();                                    //可选的
    // 数组操作
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Java 平台包含三个用于通用目的的 Set 实现：HashSet，TreeSet 及 LinkedHashSet。HashSet，将其元素存储在一个哈希表中，它具有最好的性能实现；然而它不保证迭代的顺序。TreeSet，将其元素存储在一个红黑树中，按元素的值顺序排列；本质上它比 HashSet 要慢。LinkedHashSet，是作为一个哈希表实现的，用链表连接这些元素，按元素的插入顺序排列。

下面是一个简单却有用的 Set 的习惯用法。假设存在有一个集合 c，现在想创建另外一个集合，其包含的元素与 c 相同，但消除了所有重复的元素。使用下面的代码实现了这一功能。

```
Collection<Type> noDups = new HashSet<Type>(c);
```

这行代码通过创建一个 Set（按定义不能包含重复的元素），初始化包含 c 中的所有元素。下面是这个用法的一个变化，使用 LinkedHashSet，既移除重复的元素，又保持原来集合的顺序不变。



```
Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

下面是封装了前述用法的泛型方法，返回一个 Set，此 Set 与传递进来的参数集合有相同的泛型类型。

```
public static <E> Set<E> removeDups(Collection<E> c){
    return new LinkedHashSet<E>(c);
}
```

12.4.2　Set 接口的基本操作

在 Set 接口中没有增加新的方法，它只包含从 Collection 接口继承过来的方法，包括其中 6 个基本操作方法。其中：

- size()方法返回 Set 中元素的数量（其基数）。
- isEmpty()方法如其名所示，判断 Set 是否为空。
- add()方法添加指定的元素到 Set 中（如果 Set 中还没有这个元素的话），并返回一个 boolean 值，以说明这个元素是否已经被添加了。
- 相似地，remove()方法从 Set 中移除指定的元素（如果存在的话），并返回一个 boolean 值，以说明这个元素是否已经被移除了。
- Iterator()方法返回 Set 的迭代器 Iterator。

下面通过一个示例程序说明 Set 接口的基本操作。

【例 12.1】编写程序，获取命令行参数中的字符串列表，输出其中重复的单词、不重复的单词的数量，以及消除重复以后的单词列表。实现步骤如下。

（1）编写类 FindDups，在类中创建 HashSet 的对象，并循环添加和输出命令行参数字符串。

```
import java.util.*;
public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>(); //声明不能具有重复元素的 HashSet 对象
        for (String a : args)                  //遍历命令行参数字符串数组
            if (!s.add(a))    //如果添加字符串 a 不成功，说明 s 中已经有了相同元素
                System.out.println("重复的元素: " + a);
        System.out.println(s.size() + "个单独的单词: " + s);
    }
}
```

（2）编译并运行该程序。

```
java FindDups.java
java FindDups i came i saw i left
```

（3）输出结果如下所示。

```
重复的元素: i
重复的元素: i
4 个单独的单词: [i, left, saw, came]
```

注意，这里的代码总是按 Collection 的接口类型 Set 来引用集合，而不是按 Collection 的实现类型 HashSet 来引用。强烈建议这样来写，因为这样可以带来更大的灵活性，只需要通过改变构造器就可以改变实现。如果用于保存一个集合的变量或者用于传递集合的参

数被声明为 Collection 的实现类型而不是声明为接口类型，那么为了改变其实现类型，所有这些变量和参数也必须被改变。

此外，只通过集合的接口引用集合可以防止程序员使用任何不规范的操作。在上述示例程序中，Set 的实现类型是 HashSet，它与 Set 一样，不保证其元素的顺序。如果想让程序按字母顺序输出单词列表，只有将 Set 的实现类型从 HashSet 改为 TreeSet。只在程序中改动一行的代码，那么上述程序在命令行中生成如下的输出：

```
java FindDups i came i saw i left
重复的元素: i
重复的元素: i
4 个单独的单词: [came, i, left, saw]
```

12.4.3 Set 接口的批量操作

批量操作特别适合于 Set 接口。当应用时，它们执行标准的集合代数运算。假设 s1 和 s2 是 set 集合，那么可以执行的批量操作如下所示。

- s1.containsAll(s2)：如果 s2 是 s1 的一个子集合，返回 true（如果集合 s1 包含 s2 中的所有元素，那么 s2 是 s1 的一个子集合）。
- s1.addAll(s2)：将 s1 变换为 s1 和 s2 的并集（两个集合的并集是包含两个集合的所有元素的集合）。
- s1.retainAll(s2)：将 s1 变换为 s1 和 s2 的交集（两个集合的交集指的是这个集合只包含两个集合共同的元素）。
- s1.removeAll(s2)：将 s1 不对称地变换为 s1 和 s2 的差集（例如，s1 减去 s2 的差集指的是包含所有只在 s1 中存在的元素但不在 s2 中存在的元素的集合）。

要对联合、交集或者差集进行非破坏性的计算（不修改两个集合），调用者必须在调用合适的批操作前复制一个集合。下面是据此所编写的代码：

```
Set<Type> union = new HashSet<Type>(s1); //声明 HashSet 实例对象，使用泛型版本
union.addAll(s2);                 //向 HashSet 对象中添加集合元素 s2，结果是并集
Set<Type> intersection = new HashSet<Type>(s1);
                                  //声明 HashSet 实例对象，使用泛型版本
intersection.retainAll(s2);       //获得两个集合的交集
Set<Type> difference = new HashSet<Type>(s1);//声明 HashSet 实例对象，使用泛型版本
difference.removeAll(s2);         //获得两个集合的差集
```

在上面的用法中，Set 所实现的结果类型是 HashSet，HashSet 是 Java 平台中最全面的 Set 实现。重新看一下 FindDups 程序。假设想知道参数列表中哪个单词只出现一次及哪些单词出现超过一次，但是又不想重复地输出重复的单词，可以通过创建两个 Set 来达到目的：一个包含参数列表中的每一个单词，另一个只包含重复的单词。只出现一次的单词是这两个集合的差。

【例 12.2】使用 HashSet 进行单词统计应用程序示例：

```
import java.util.*;
public class FindDups2 {
    public static void main(String[] args) {
        Set<String> uniques = new HashSet<String>();
                                          //声明 HashSet 实例对象，使用泛型版本
```



```
        Set<String> dups = new HashSet<String>();
                                          //声明 HashSet 实例对象，使用泛型版本
        for (String a : args)             //遍历命令行参数字符串数组
            if (!uniques.add(a))          //如果 a 已经存在
                dups.add(a);              //那么，将 a 添加到 dups 集合中
        // 破坏性的集合差
        uniques.removeAll(dups);          //从 uniques 中移除所有具有重复的单词
        System.out.println("不重复的单词: " + uniques);
        System.out.println("重复的单词  : " + dups);
    }
}
```

编译并运行上述程序，使用相同的命令行参数列表（i came i saw i left），程序产生下面的输出结果：

```
不重复的单词: [left, saw, came]
重复的单词  : [i]
```

没有共同元素的集合代数运算是对称集合差（symmetric set difference）。对称集合差是这样的集合，它只包含或者属于这个集合的元素，或者属于另一个集合的元素，但不同时属于两个集合的元素。下面的代码非破坏性地计算两个集合的对称集合差：

```
Set<Type> symmetricDiff = new HashSet<Type>(s1);
symmetricDiff.addAll(s2);
Set<Type> tmp = new HashSet<Type>(s1);
tmp.retainAll(s2));
symmetricDiff.removeAll(tmp);
```

12.4.4　Set 接口的数组操作

Set 接口的数组操作与任何其他的 Collection 接口都是一样的，没有任何特殊的地方，在此不再重复。

12.5　List 接口

List 是一个有序的集合（有时被称为序列）。List 可以包含重复的元素。除了从 Collection 继承过来的操作之外，List 接口还包括以下的操作。

- 按位置访问：根据元素在序列中的位置索引访问元素。
- 查找：在序列中查找指定的对象，并返回其位置索引。
- 迭代：扩展了 Iterator 接口，以利用序列的顺序特性。
- List 子集合：在序列上执行任意范围的操作。

12.5.1　List 接口的定义

List 接口定义如下。

```
public interface List<E> extends Collection<E> {
    // 按位置访问
    E get(int index);
    E set(int index, E element);                          //可选地
    boolean add(E element);                               //可选地
```

```
    void add(int index, E element);                              //可选地
    E remove(int index);                                         //可选地
    boolean addAll(int index,Collection<? extends E> c);         //可选地
    // 查找
    int indexOf(Object o);
    int lastIndexOf(Object o);
    // 迭代
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    // 子集合
    List<E> subList(int from, int to);
}
```

Java 平台包含两个通用的 List 实现：

- ArrayList，通常有较好的性能实现；
- LinkedList，在一定条件下提供较好的执行性能。

另外，Vector 已经重新实现了 List。

12.5.2 List 接口与 Vector 的比较

List 和 Vector 之间有许多相似之处（当然了，List 是个接口，而 Vector 是一个具体的实现）。List 对 Vector 的一些常用的 API 进行了改进，使之更加简单易用。Vector 的常用操作，如 elementAt()和 setElementAt()，它们都具有简洁的名称。考虑如下的赋值语句：

```
a[i] = a[j].times(a[k]);
```

使用 Vector 的等价操作是：

```
v.setElementAt(v.elementAt(j).times(v.elementAt(k)), i);
```

使用 List 的等价操作是：

```
v.set(i, v.get(j).times(v.get(k)));
```

可能读者已经注意到了，在 List 中用 set 方法代替了 Vector 的方法 setElementAt()，并反转了参数的顺序，这样与数组的操作相匹配。考虑下面的赋值语句：

```
gift[5] = "golden rings";
```

使用 Vector 的等价操作是：

```
gift.setElementAt("golden rings", 5);
```

使用 List 的等价操作是：

```
gift.set(5, "golden rings");
```

从一致性上考虑，List 的方法 add（int, E）代替了 Vector 的 insertElementAt（Object,int）操作，并反转了参数的顺序。在 Vector 中的三个对于范围的操作（indexOf(),lastIndexOf(), setSize()），已经被 List 中的 subList()操作统一代替，这个方法更加强大，包容性更强。可以看到，List 的元素操作比 Vector 的更加简洁。

12.5.3 从 Collection 继承的操作

List 接口具有从 Collection 接口继承过来的所有操作的功能。例如，remove()操作总是从 list 中移除第一个与指定元素相匹配的元素。Add()和 addAll()操作总是将新的元素追加



到序列的尾部。因此，下面的用法可以将一个 List 对象与另一个 List 对象连接。

```
list1.addAll(list2);
```

下面是一个这种用法的非破坏性的形式，创建第三个 List 对象，由第二个 List 对象追加到第一个 List 对象上组成。

```
List<Type> list3 = new ArrayList<Type>(list1);
list3.addAll(list2);
```

注意，在这种非破坏性形式的用法中，利用了 ArrayList 的标准转换构造器。和 Set 接口一样，List 接口增强了两个函数：equals() 和 hashCode()，因而两个 List 对象可以进行逻辑相等性的比较，而不用考虑它们的实现类。如果两个 List 对象包含相同的元素并有相同的顺序，那么它们是相等的。

12.5.4　按位置访问和查找操作

基本的按位置访问的操作有 get()、set()、add()和 remove()方法，它们的功能和在 Vector（向量）中相对应的方法（elementAt()、setElementAt()、insertElementAt()和 removeElementAt()方法）很类似，只有一点需要注意的差异：set()和 remove()方法返回元素被改写或被移除前的值；而 Vector 中相应的方法（setElementAt()和 removeElementAt()）什么也不返回（void）。查找操作 indexOf()和 lastIndexOf()功能与 Vector 中相同名称的操作相同。

addAll()方法将指定集合的所有元素插入到指定位置。元素按集合的迭代器返回的顺序被插入 list。这种调用是位置访问，与 Collection 的 addAll()操作类似。下面是一个简单的方法，用来交换 List 对象中两个不同索引处的值。

```
public static <E> void swap(List<E> a, int i, int j){
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

这是一个多态的算法：交换任何 List 中的两个元素，而不必考虑其实现类型。下面是使用前面的 swap()方法的另外一个多态算法。

```
public static void shuffle(List<?> list, Random rnd) {
   for (int i = list.size(); i > 1; i--)
      swap(list, i - 1, rnd.nextInt(i));
}
```

这个算法是包含在 Java 平台 Collections 类中的一个算法，它使用指定的随机种子随机地安排指定列表的序列。这个方法的实现是从 List 对象的底部开始往上，重复地将随机选择的元素与当前位置的元素交换。这种方式严格地要求 list.size()-1 次交换。

【例 12.3】使用 shuffle()算法随机地输出其参数列表中的单词。

```
import java.util.*;
public class ShuffleDemo1 {
   public static void main(String[] args) {
      List<String> list = new ArrayList<String>();//创建一个动态数组对象
      for (String a : args)                    //循环遍历命令行参数中的每一个元素
         list.add(a);                          //将元素添加到数组中
      Collections.shuffle(list, new Random());
                          //使用 Collections 类的 shuffle 方法对 list 随机排序
```

```
        System.out.println(list);
    }
}
```

实际上，这个方法可以更简洁、更快速。在 Arrays 类中有一个静态的工厂方法，叫 asList()，将一个数组当做一个 List 看待。这个方法并不复制数组。在 List 中的改变也同样反应在数组当中，反之亦然。最终的 List 不是一个通用的 List 实现，因为它并没有实现 add()和 remove()操作：数组是不可改变大小的。使用 Arrays.asList()并调用方法 shuffle()（使用一个默认的随机种子），可以得到以下的精简的程序，其作用与上面的程序相同。

【例 12.4】使用 shuffle()算法随机地输出其参数列表中的单词：

```
import java.util.*;
public class ShuffleDemo2 {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);     //创建一个动态数组对象
        Collections.shuffle(list);//使用Collections类的shuffle方法对list随机排序
        System.out.println(list);            //输出随机排序以后的 list 中的元素
    }
}
```

先编译 ShuffleDemo2.java，然后在命令行下，使用如下的命令运行，其输出结果中字符串的顺序与命令行中输入的顺序是不同的。

```
>java ShuffleDemo2 hello every one
[every, one, hello]
```

12.5.5 List 迭代器

由 List 的 iterator()方法所返回的 Iterator 迭代器，以固有顺序返回 List 对象中的元素。List 还提供一个更加强大的迭代器，称为 ListIterator，它允许从前到后地遍历 List 对象，也可以从后到前地遍历，在迭代期间修改 List 对象，以及获得迭代器的当前位置。ListIterator 接口如下所示。

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove();                  //可选地
    void set(E e);                  //可选地
    void add(E e);                  //可选地
}
```

该接口中有三个方法继承自 Iterator（hasNext()，next()和 remove()），并且实现同样的功能。hasPrevious()和 previous()方法使用方式和 hasNext()和 next()类似。前者操作指向隐含游标前的元素，反之后者指向隐含游标后面的元素。previous()操作向后移动游标，而 next()向前移动游标。下面是向后迭代一个 List 对象的规范用法：

```
for(ListIterator<Type> it = list.listIterator(list.size());it.hasPrevious(); ){
    Type t = it.previous();
    …
}
```



注意上面的用法当中 listIterator 的参数。List 接口有两个形式的 listIterator()方法。不带参数的形式返回一个位于 List 对象起始位置的 ListIterator；带有一个 int 类型参数的形式返回一个位于 List 对象指定索引处的 ListIterator。初始调用 next()方法，将返回索引处的元素。对 previous()的初始调用将返回索引为 index-1 的元素。在一个长度为 *n* 的 List 对象中，有 *n*+1 个可用的 index 索引值，包括从 0 到 *n*。

直观地说，游标总是位于两个元素之间——通过调用 previous()将要返回的元素和通过调用 next()将要返回的元素。*n*+1 个索引值对应于元素间 *n*+1 个间隙，从第一个元素前到最后一个元素后。图 12.2 显示了一个包含有 4 个元素的 List 对象的 5 个可能的游标位置。



图 12.2　五个可能的游标位置

可以混合调用 previous()和 next()，但是一定要小心。第一个对 previous()的调用返回的元素与最后一个对 next()的调用返回的元素相同。同样，在一系列地调用 previous()以后，第一个对 next()的调用返回的元素与最后一个对 previous()的调用返回的元素相同。

nextIndex()方法返回通过调用 nex()t 方法所返回的元素的索引，previousIndex()方法返回通过调用 previous()方法所返回的元素的索引。这些调用通常被用于查找某些元素的位置或者记录 ListIterator 的位置，这样另外一个具有相同位置的 ListIterator 就可以被创建。由 nextIndex()返回的数字总是比 previousIndex()返回的数字大 1。这意味着两种截然不同的行为：

- 当游标位于第一个元素之前时，对 previousIndex()的调用返回-1；
- 当游标位于最后一个元素之后时，对 nextIndex()的调用返回 list.size()。

下面是对 List.indexOf()的一个可能的实现，具体地用到了这些方法。

```
public int indexOf(E e) {                    //泛型方法，查找集合中元素 e 的索引
    for (ListIterator<E> it = listIterator(); it.hasNext(); )
        if (e == null ? it.next() == null : e.equals(it.next()))
            return it.previousIndex();
    return -1;                               // 没有找到元素，返回-1
}
```

值得注意的是，即使是向前遍历整个 List 对象，indexOf()方法仍然返回 it.previousIndex()。因为 it.nextIndex()返回将要进行检查的元素的索引，而现在是需要返回正在检查的元素的索引。

Iterator 接口提供了 remove()操作，用来移除使用 next()方法从集合当中最后返回的元素。对于 ListIterator，同样的操作移除由 next()或 previous()所返回的最后的元素。ListIterator 接口还提供了两个额外的操作来修改 List 对象——set()和 add()。其中 set()方法使用指定的元素覆盖了由 next()或 previous()方法返回的最后的元素。在下面的多态算法中，调用 set()方法来使一个新值替换集合中某一个指定值：

```
public static <E> void replace(List<E> list, E val, E newVal) {
    for (ListIterator<E> it = list.listIterator(); it.hasNext(); )
        if (val == null ? it.next() == null : val.equals(it.next()))
                                         //如果集合中下一个值与 val 值相等或为 null
            it.set(newVal);              //使用新的值替换集合与 val 相同的值或 null
}
```

在上面这个例子中，唯一有点微妙的是 val 和 it.next()的相等测试。需要特别指明 val

的 null 值以防止出现 NullPointerException 异常。

add()方法将一个新的元素插入到 List 对象中位于当前游标位置之前。下面的多态算法的例子当中，使用一个指定的 List 对象所包含的序列值来替换所有指定的值，在这中间用到了 add()方法。

```
public static <E> void replace(List<E> list, E val,
                               List<? extends E> newVals) {
    for (ListIterator<E> it = list.listIterator(); it.hasNext(); ){
        if (val == null ? it.next() == null : val.equals(it.next())) {
            it.remove();                          //移除 null 空值或与 val 相等的值
            for (E e : newVals)
                it.add(e);
        }
    }
}
```

12.5.6 子集合操作

子集合操作方法“subList(int fromIndex, int toIndex)”返回当前 List 对象的某个范围的子集合，其索引范围从 fromIndex(包括 fromIndex)到 toIndex（不包括 toIndex）。这个“半开区间”类似于 for 循环：

```
for (int i = fromIndex; i < toIndex; i++) {
    ...
}
```

使用 subList()方法所返回的子集合是调用该方法的集合的一个视图，它由调用该方法的 List 对象所支持，因此在子集合中的改变将会反映到集合上。

这个方法消除了对于显式子集合操作（通常存在于数组中的排序）的需要。任何希望一个 List 对象能被作为子集合的操作，都通过传递一个 subList 子集合实现，而不必使用整个 List。例如，在下面的代码中，从一个 List 中移除一个子集合的元素：

```
list.subList(fromIndex, toIndex).clear();
```

相似的用法也可以用于在一定的范围内查找一个元素：

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

注意上述用法中，返回的是在子集合 subList 中找到的元素的索引，而不是其背后的 List 的索引。任何在 List 上的多态的算法，如 replace()和 shuffle()示例，在由 subList 返回的子集合中同样可以使用。

下面是一个多态的算法，其实现使用 subList 来处理一副牌中的一手牌。处理过程是这样的：返回一个新的 List（一手牌），包含来自指定的 List（一副牌）的尾部的元素数量。所返回的一手牌中的元素从一副牌中被移除。

```
public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();                          //指定一手牌的大小
    List<E> handView = deck.subList(deckSize - n, deckSize);//获得一手牌
    List<E> hand = new ArrayList<E>(handView);
    handView.clear();
    return hand;
}
```



注意在这个算法实现中，从一副牌的尾部移除一手牌。对许多普通的 List 实现来说，如 ArrayList，从 List 对象的尾部移除元素的性能，要远好于从起始部分移除元素的性能。

【例 12.5】编写一个程序，模拟扑克牌游戏，结合使用 dealHand()方法和 Collections.shuffle()方法来从常见的 52 张牌中生成一手牌。程序需要两个命令行参数：

- 要处理几手牌；
- 每手牌中的卡片数量。

（1）编写程序代码如下。

```
import java.util.*;
class Deal {
   public static void main(String[] args) {
      int numHands = Integer.parseInt(args[0]);//声明整型变量，代表玩家数量
      int cardsPerHand = Integer.parseInt(args[1]);
                                              //声明整型变量，代表每手牌的数量
      // 构建一副含 52 张卡片的牌.
      String[] suit = new String[]{"黑桃", "红心", "方块", "梅花"};
                                              //代表花色的数组
      String[] rank = new String[]            //代表大小的数组
         {"A","2","3","4","5","6","7","8","9","10","J","Q","K"};
      List<String> deck = new ArrayList<String>();
      for (int i = 0; i < suit.length; i++)//组合出每线牌的类型：花色+大小
         for (int j = 0; j < rank.length; j++)
            deck.add(suit[i] + rank[j]);
      Collections.shuffle(deck);              //对扑克牌随机排序（相当于洗牌）
      for (int i=0; i < numHands; i++)        //调用 dealHand 方法生成 4 手牌
         System.out.println(dealHand(deck, cardsPerHand));
   }
   //生成一手牌的方法
   public static <E> List<E> dealHand(List<E> deck, int n) {
        int deckSize = deck.size();
        List<E> handView = deck.subList(deckSize - n, deckSize);
        List<E> hand = new ArrayList<E>(handView);
        handView.clear();
        return hand;
   }
}
```

（2）使用如下的命令编译并运行上述程序，注意在运行时传递进两个参数。

```
javac Deal.java
java Deal 4 5
```

（3）程序输出结果为：

```
[红心 8,黑桃 J,黑桃 3,黑桃 4,方块 K]
[方块 4, 梅花 A,梅花 6,红心 J,红心 Q]
[黑桃 7,黑桃 5,方块 2,方块 Q,梅花 9]
[黑桃 8,方块 6,黑桃 A,红心 3,红心 A]
```

虽然 subList 操作功能非常地强大，在使用它的时候仍然要很小心。如果以任何其他途径而不是经由返回的子集合从其支持的 List 中添加或移除元素，那么由 subList 返回的子集合会变得很不确定。因此，强烈建议将 subList 返回的子集合作为瞬时对象使用——在

其背后的 List 上执行一个或一系列子集合操作。需要注意的是，修改一个子集合的子集合并继续使用原始的子集合（虽然不是并发的）是合法的。

12.6 Queue 接口

Queue（队列）是一个具有元素片优先级的集合。除了基本的 Collection 的操作之外，队列提供了额外的插入、删除和检查元素的操作。

12.6.1 Queue 接口的定义

在 Queue 接口中声明了一系列的方法，凡是实现 Queue 接口的类都要实现这些方法。所以先了解 Queue 接口的定义，掌握其声明方法的含义和用法，有助于快速且全面地理解该接口的用法。Queue 接口的定义如下。

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

12.6.2 Queue 接口的操作

每一个 Queue 的方法都以两种形式存在：一种是如果操作失败抛出一个异常，另一种是如果操作失败返回一个特定的值（可以是 null 或 false，根据操作的不同）。Queue 接口常用的结构如表 12.1 所示。

表 12.1 队列接口结构

	抛出异常	返回特定的值
插入	add(e)	offer(e)
删除	remove()	poll()
检查	element()	peek()

队列通常以 FIFO（先进先出）的形式排序元素，但也有例外的情况，如优先级队列，它根据元素的值排列元素。不管哪种顺序，调用 remove()或 poll()方法会删除队列的头元素。在一个 FIFO 队列中，所有新的元素都插入到队列的尾部。

还有一种队列是“有限制队列（bounded）”，它可以限制 Queue 所持有的元素数量。在 java.util.concurrent 中的一些 Queue 接口的实现就是有限制的，但在 java.util 中的实现则是无限制的。

Queue 从 Collection 中继承的 add()方法用来插入一个元素。如果插入以后队列溢出，就会抛出一个 IllegalStateException 异常。专门用于有限制队列的 offer()方法，与 add()方法唯一的不同在于当插入失败时返回 false。

remove()和 poll()方法都删除并返回队列的头元素。在队列为空的情况下，remove()方法抛出 NoSuchElementException 异常，而 poll()返回 null。



element()和 peek()方法返回但不删除队列的头元素。它们之间唯一的不同在于如果队列是空的，element()抛出 NoSuchElementException 异常，而 peek()返回 null。

Queue 的实现一般不允许插入 null 元素。而 LinkedList 实现（是另外一种对 Queue 的实现）允许 null 元素。但应该尽量避免使用 null 元素，因为 null 用做 poll()和 peek 方法()的特殊的返回值。Queue 实现通常不重新定义 equals()和 hashCode()方法。

12.6.3　Queue 接口应用

在下面的示例程序中，一个队列被用于实现一个倒数计数器。该队列预先加载从命令行指定的一个数值到 0 之间的所有整数值，按降序排列。然后从队列中删除数值并每秒输出一个。

【例 12.6】使用队列 Queue 接口的倒数计数器。

```
import java.util.*;
public class Countdown {
    public static void main(String[] args) throws InterruptedException {
        int time = Integer.parseInt(args[0]);//获得命令行参数并转换为 int 型
        Queue<Integer> queue = new LinkedList<Integer>();
                                                    //创建一个 LinkedList 对象
        for (int i = time; i >= 0; i--)             //使用 for 循环倒计时
            queue.add(i);                           //向队列中添加元素（时间 i）
        while (!queue.isEmpty()) {                  //如果队列不为空
            System.out.println(queue.remove());  //移除队列中头一个元素
            Thread.sleep(1000);                     //线程暂停执行 1 秒钟
        }
    }
}
```

在下面的示例中，使用优先级队列来排序一个元素集合：

```
static <E> List<E> heapSort(Collection<E> c) {
    Queue<E> queue = new PriorityQueue<E>(c);
                                        //创建优先级队列对象，其元素为集合 c 中的元素
    List<E> result = new ArrayList<E>();//创建一个 ArrayList 动态数组对象
    while (!queue.isEmpty())            //如果优先级队列中元素不为空
        result.add(queue.remove());     //从队列中删除元素，并添加到动态数组中
    return result;
}
```

12.7　Map 接口

Map 是一种包含键值对的元素的集合。Map 不能包含重复的键：每个键最多可映射到一个值。它是数学函数的抽象模型。

12.7.1　Map 接口的定义

在 Map 接口中声明了一系列的方法，凡是实现 Map 接口的类都要实现这些方法。所以先了解 Map 接口的定义，掌握其声明方法的含义和用法，有助于快速且全面地理解该接口的用法。Map 接口的定义如下。

```
public interface Map<K,V> {
```

```
    // 基本操作
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    // 批量操作
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    // 集合视图
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    // 用于 entrySet 元素的接口
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Java 平台包含三种通用的 Map 实现：HashMap，TreeMap 和 LinkedHashMap。它们的行为和执行性能正好与 HashSet、TreeSet 和 LinkedHashSet 类似。另外，Hashtable（哈希表）重新实现了 Map。

12.7.2 Map 接口的基本操作

Map 的基本操作（put()，get()，containsKey()，containsValue()，size()，isEmpty()）的行为非常类似于 Hashtable 中相应的操作。下面的程序生成一个在参数列表中出现的单词的频度表。在频度表中，将每一个单词映射到它在参数列表中出现的次数。

【例 12.7】编写统计命令行参数字符串出现次数的程序。

```
import java.util.*;
public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();
        // 从命令行初始化频率表
        for (String a : args) {
            Integer freq = m.get(a);      //获得键为 a 的元素的值
            m.put(a, (freq == null) ? 1 : freq + 1);
                                          //设置 a 出现的次数，保存在 Map 对象中
        }
        System.out.println(m.size() + "个不同的单词:");
        System.out.println(m);
    }
}
```

在上述程序中，需要注意的是 put 语句的第二个参数。该参数是一个条件表达式，用来设置一个单词的出现频率。如果这个单词从来没有出现过，设置为 1；如果已经出现过，设置为当前值增 1。

编译并运行此程序：

```
java Freq if it is to be it is up to me to delegate
```



程序的输出结果如下。

```
8 个不同的单词：
{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

假设想以字母顺序看到此频度表，那么所有要做的事情就是将 Map 的实现类型由 HashMap 改为 TreeMap。将上面程序中的 HashMap 改为 TreeMap，重新编译并以同样的命令行参数运行，输出结果为：

```
8 个不同的单词：
{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

同样，可以使程序按单词在命令行参数中的顺序输出频度，只需要将 map 实现的类型改为 LinkedHashMap。那么上述程序的输出结果为：

```
8 个不同的单词：
{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}
```

与 Set 和 List 接口类似，Map 也加强了对 equals()和 hashCode()方法的要求，这样两个 Map 对象可以进行逻辑相等比较，而不必考虑它们的实现类型。如果两个 Map 实例代表相同的键-值映射，那么它们就是相等的。

按惯例，所有通过的 Map 实现都提供接收一个 Map 对象作为参数的构造器，并初始化新的 Map 使其包含指定 Map 中所有的健-值对。这个标准 Map 转换构造器完全与标准 Collection 构造器相类似：允许调用者创建一个希望实现的类型的 Map，初始化包含另一个 Map 的所有的映射，而不必考虑另一个映射的实现类型。例如，假设有一个名为 m 的 Map。下面这行代码创建一个新的 HashMap，初始化包含 m 中所有的相同的键-值映射。

```
Map<K,V> copy = new HashMap<K,V>(m);
```

12.7.3　Map 接口的批量操作

Map 接口中的批量操作主要有 clear()方法、putAll()方法。其中：clear()方法从 Map 中删除所有的映射；putAll()方法是 Map 中类似于 Collection 接口的 addAll()方法。

putAll 除了将一个 Map 填充进另一个 Map 这种用法之外，还有第二种更为巧妙的用法。假设一个 Map 被用于代表一个属性-值对的集合；putAll()方法与 Map 的转换构造器结合使用，提供一种优雅的方式来实现使用默认值的属性映射。下面是使用这种技术的一个静态工厂方法的示范。

```
    static <K, V> Map<K, V> newAttributeMap(Map<K, V> defaults, Map<K, V>
overrides) {
        Map<K, V> result = new HashMap<K, V>(defaults); //创建一个新的 Map 对象
        result.putAll(overrides);                       //填充进另一个 Map 集合
        return result;                                  //返回填充后的 Map 对象
    }
```

12.7.4　集合视图

集合视图方法使得 Map 可以用集合的形式展现其元素。Map 的集合视图有如下三种形式，分别用来获得键的集合、值的集合及键值对的集合：

- keySet——Map 中包含的键的 Set 集合。

- values——Map 中包含的值的 Collection 集合。这个集合不是一个 Set，因为多个键可以映射到同一个值。
- entrySet——Map 中包含的键-值对的 Set 集合。在这个集合中的元素的类型是 Map.Entry，它是 Map 接口提供了一个小的嵌套的接口。

集合视图提供了遍历一个 Map 的唯一的方式。下面的示例代码演示了使用一个 for-each 结构遍历一个 Map 的键的标准用法。

```
for(KeyType key : m.keySet())
    System.out.println(key);
```

以及使用 iterator 迭代器遍历一个 Map 的键的标准用法：

```
// 基于键的某些属性过滤一个 map.
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
   if (it.next().isBogus())
      it.remove();
```

迭代值的用法与此类似。下面是迭代键-值对的用法：

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())
   System.out.println(e.getKey() + ": " + e.getValue());
```

起初，很多人担心这种用法会使程序变慢，因为每次调用一个集合视图时，Map 不得不创建一个新的集合实例。不过不用担心的是：每次调用一个集合视图时，一个 Map 可以总是返回同样的对象。在 java.util 中所有的 Map 实现都是这么做的。

使用三种集合视图，调用一个 Iterator 迭代器的 remove()操作，会删除其背后 Map 中相关联的条目（首先假设其背后的 Map 支持元素删除）。这在上面的过滤用法中已经示范过。

对于 entrySet 视图，在迭代期间通过调用一个 Map.Entry 的 setValue()方法，有可能改变与一个键相关联的值（首先也假设 Map 支持值的修改）。注意这是在迭代期间修改 Map 的唯一安全的方式；如果在迭代处理过程中以任何其他的方式修改 Map，其表现行为总是不确定的。

集合视图以多种形式支持元素删除——remove()，removeAll()，retainAll()和 clear()操作，以及 Iterator.remove()操作。集合视图不支持在任何情况下的元素添加，因为这将使得 keySet 和 values 视图毫无意义，并且这对于 entrySet 视图是不必要的，因为其背后的 Map 的 put()和 putAll()方法提供了相同的功能。

12.8 SortedSet 接口

一个 SortedSet 是按升序排列其元素的 Set 集合，根据元素的自然顺序或根据 SortedSet 创建时提供的 Comparator（比较器）进行排序。除了正常的 Set 操作之外，SortedSet 接口还提供了以下的操作：

- 子集合——允许在排序的 Set 集合上的任意的子集合操作。
- 端点访问——返回排序的 Set 集合中第一个或最后一个元素。
- 比较器访问——返回用于对 Set 进行排序的 Comparator。

12.8.1 SortedSet 接口的定义

在 SortedSet 接口中声明了一系列的方法，凡是实现 SortedSet 接口的类都要实现这些



方法。所以先了解 SortedSet 接口的定义，掌握其声明方法的含义和用法，有助于快速且全面地理解该接口的用法。SortedSet 接口的定义如下。

```
public interface SortedSet<E> extends Set<E> {
    // 子集合
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    // 端点访问
    E first();
    E last();
    // 比较器访问
    Comparator<? super E> comparator();
}
```

12.8.2　SortedSet 接口的操作

SortedSet 从 Set 继承过来的操作，其功能在 SortedSet 集合上和一般的 Set 集合上是相同的，除了两点不同：

- 由 iterator()操作返回的迭代器 Iterator 按排列顺序遍历 SortedSet 集合。
- 由 toArray()方法返回的数组有序地包含 SortedSet 集合的元素。

Java 的 SortedSet 实现的 toString()方法返回包含 SortedSet 中的所有元素，按顺序排列。

12.8.3　SortedSet 接口的标准构造器

按惯例，所有通用的 Collection 实现提供一个标准构造器，这个标准构造器接收一个集合参数；同样地，SortedSet 也不例外。

在 TreeSet 中，这个构造器创建一个实例，根据元素的自然顺序对元素排序。另外，TreeSet 还提供一个接收 SortedSet 的构造器，并返回一个包含同样的按相同标准排序的元素的新的 TreeSet。注意，是参数的编译时类型而不是其运行时类型来决定这两个构造器中的哪一个被调用（以及是否保持排序的标准）。

SortedSet 的实现还按惯例提供一个这样的构造器，它接收一个 Comparator 比较器并返回一个空的按指定的 Comparator 排序的 Set。如果将 null 传递给此构造器，将返回一个元素按自然顺序排列的 Set。

12.8.4　子集合操作

SortedSet 的子集合的操作，与 List 接口提供的子集合操作有一些类似，但也有很大的不同。一个 SortedSet 的子集合甚至其背后的排序的 Set 被直接修改后也继续保持有效。一个 SortedSet 的子集合真正地是位于元素空间指定部分的一个 Set 区间的窗口。对子集合的改变会写回其背后的 SortedSet，反过来也一样。因此，在 SortedSet 上长期使用子集合视图是没有问题的，而不像 List 上的子集合。

SortedSet 提供三种子集合的操作：subset()、headSet()和 tailSet()。subset()方法接收两个端点，就像 subList()一样。这里的端点是对象而不使用索引，并且在 SortedSet 中的元素必须是可以使用 Set 的 Comparator 或元素的自然顺序进行比较的。像 subList()一样，这个

区间是半开的，包括低的一端的端点，但不包括高的一端的端点。

因此，下面的一行代码用来统计在一个称为 dictionary 的 String 类型的 SortedSet 中，在“doorbell”和“pickle”中间有多少个单词，包括“doorbell”但是不包括“pickle”。

```
int count = dictionary.subSet("doorbell","pickle").size();
```

以相同的方式，下面的一行代码调用 clear()方法，删除所有以字母 f 开头的元素。

```
dictionary.subSet("f","g").clear();
```

类似的技巧可被用于统计以每一个字母开始的单词有多少个，如下的代码所示。

```
for (char ch = 'a'; ch <= 'z'; ) {      //遍历字母'a'和'z'之间的单词
   String from = String.valueOf(ch++); //获取字母 ch 的下一个字母的字符串表示
   String to = String.valueOf(ch);     //获取字母 ch 的字符串表示
   System.out.println(from + ": " +
      dictionary.subSet(from, to).size());//获取 from 和 to 之间的子集合的大小
}
```

假设想要一个“封闭的区间”，包含两个端点，而不是开放的区间。下面的代码统计在字典中在“doorbell”和“pickle”中间有多少个单词，包括 doorbell 和 pickle。

```
count = dictionary.subSet("doorbell","pickle\0").size();
```

相似的技巧可用于一个“开区间”，即两个端点都不包括。下面的代码统计“doorbell\0”和“pickle”之间的单词数量，不包括两个端点。

```
count = dictionary.subSet("doorbell\0","pickle").size();
```

除了 subSet 之外，SortedSet 接口还包括另外两个区间范围的操作：headSet()和 tailSet()，都接收一个单一的 Object 参数。前者返回 SortedSet 的起始部分的视图，直到但不包括指定的对象。后者返回 SortedSet 的结尾部分的视图，以指定的对象起始一直到 SortedSet 的结束。因此，下面的代码将字典分为两个不相连的卷（a-m 和 n-z）。

```
SortedSet<String> volume1 = dictionary.headSet("n");
SortedSet<String> volume2 = dictionary.tailSet("n");
```

12.8.5 端点操作

SortedSet 接口包含有返回有序 Set 集合中第一个和最后一个元素的操作，分别称为 first()和 last()。经常使用 SortedSet 的一种方式可能是进入到 Set 内部，然后往前或往后遍历。从内部很容易实现向前遍历：获得一个 tailSet，然后遍历它。但是很不幸的是，向后遍历不容易实现。下面的代码获得元素空间中一个小于指定的对象 o 的第一个元素。

```
Object predecessor = ss.headSet(o).last();
```

这是从一个 SortedSet 内部开始从一个端点向后获取一个元素的很好的途径。它可以被重复应用以向后迭代，但是效率很低，因为这样会要求查找每一个返回的元素。

12.9 SortedMap 接口

一个 SortedMap 是一个其所含条目按升序排列的 Map，根据键的自然顺序，或者根据创建 SortedMap 时提供的 Comparator 排序。除了正常的 Map 的操作之外，SortedMap 还提



供有如下的操作：

- 子集合——允许在排序的 Map 集合上的任意的子集合操作。
- 端点访问——返回排序的 Map 集合中第一个或最后一个键。
- 比较器访问——返回用于对 Map 进行排序的 Comparator。

12.9.1　SortedMap 接口的定义

在 SortedMap 接口中声明了一系列的方法，凡是实现 SortedMap 接口的类都要实现这些方法。所以先了解 SortedMap 接口的定义，掌握其声明方法的含义和用法，有助于快速且全面地理解该接口的用法。下面是 SortedMap 接口的定义：

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

12.9.2　SortedMap 接口的操作

SortedMap 中从 Map 继承来的操作从功能上来说，与正常的 Map 中的操作是相同的，除了以下两点：

- 在任何 SortedMap 的集合视图上的 iterator()方法返回的迭代器 Iterator 按顺序遍历集合。
- 由集合视图的 toArray()方法返回的数组包含有序的键、值或条目。

所有 Java 平台的 SortedMap 实现的集合视图的 toString()方法都返回一个字符串，有序地包含视图的所有元素。

12.9.3　SortedMap 接口的标准构造器

按惯例，所有通用的 Map 实现提供一个接收 Map 参数的标准转换构造器；SortedMap 也不例外。在 TreeMap 中，此构造器创建一个实例，实例根据其键的自然顺序排序其条目。TreeMap 还提供一个接收一个 SortedMap 参数的构造器，并返回一个包含同样映射的新的 TreeMap，按同样的规则排序。

注意，是参数的编译时类型，而不是它的运行时类型，决定了在优先选择普通的 Map 构造器时是否调用 SortedMap 构造器。

按惯例，SortedMap 的实现还提供一个接收 Comparator 参数的构造器，并返回一个根据指定的 Comparator 排序的空的 Map。如果给构造器传递一个 null，则返回一个根据键的自然顺序对映射进行排序的 Map。

12.9.4　与 SortedSet 接口的比较

因为这个接口是与 SortedSet 类似的一个 Map，所以在“SortedSet 接口”一节中讲到的用法和代码示例只需要一点修改就都可用于 SortedMap。



12.10 实现

实现是用于存储集合的数据对象，实现了在“接口”一节所描述的那些接口。通俗地说，实现是指这样一些类，这些类实现了上一节所介绍的各种集合框架中的接口。在实际开发程序时，对集合的使用主要是使用这些实现了集合接口的集合类。

12.10.1 实现的类型

对于集合框架中集合接口的实现，主要包括如下几种类型。

- 通用目的的实现：是最经常使用的实现，是为日常使用而实现的。
- 特殊目的的实现：被设计用于特殊情况，并显示了非标准的执行特征、使用约束或行为。
- 并发实现：被设计用于高并发性，特别是单线程的开销情况下。这些实现是 java.util.concurrent 包的一部分。
- 其他实现：用于与其他特定目的的实现。

用于通用目的的实现总结在表 12.2 中。

表 12.2 通用目的实现

接口	实现				
	哈希表	可变数组	树	链表	哈希表+链表
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

如表 12.2 所示，Java 集合框架提供了数个对 Set、List 和 Map 接口的通用实现。注意 SortedSet 和 SortedMap 接口并不在表中。这些接口中的每一个都有一个实现（TreeSet 和 TreeMap），在 Set 和 Map 列列出。有两个通用的对 Queue 的实现：LinkedList，它同时还是 List 的实现；以及 PriorityQueue（它没有在表中出现）。这两个实现提供了不同的语义：LinkedList 是 FIFO 的，而 PriorityQueue 根据元素的值排序。

每一个通用目的的实现都提供了所有其接口当中包含的操作，都允许 null 元素、键和值，都不是同步的（线程安全的），都能在迭代期间检测非法的并发修改，并迅速失败并清除，都是可序列化的并且都支持公共的 clone()方法。

事实上，这些实现都是非同步的，这与以前旧版本的实现是不同的：遗留下来的集合 Vector 和 Hashtable 是同步的。之所以这样改变，是因为当集合被频繁地使用时，同步是无益的。这样的使用包括单线程使用、只读使用，以及作为大的数据对象的一部分使用（该大数据对象有自己的同步）。一般来说，这是一个好的 API 设计的实践，不使用户为他们不使用的特性而付出额外的开销。换句话说，不必要的同步在一定情况下会导致死锁。

如果需要线程安全的集合，可以使用同步包装器，允许任何集合被转换为一个同步的集合。此外，java.util.concurrent 包提供 BlockingQueue 接口的同步实现，该接口扩展了 Queue，以及对 ConcurrentMap 接口的实现，该接口扩展自 Map。这些实现比仅同步实现提供更高的并发性。



作为一个原则，在使用集合时，应该考虑的是接口，而不是实现。这就是为什么在本节中没有程序示例的原因。大多数情况下，对于实现的选择，只影响执行性能。当创建一个接口并立即赋予一个新的集合给相对应的接口类型的变量时，选择实现（或者传递集合到一个需要接口类型参数的方法中）。以这种方式，程序并不依赖于一个给定实现当中的任何增加的方法，这样程序可以在任何时候因性能原因或行为细节而改变实现。

12.10.2　Set 接口的实现

从上一节中可知，Set 接口中的元素不允许有重复的元素。因此实现 Set 接口的实现类要遵循这样的原则。Set 实现分为通用实现和特殊实现。

1．通用 Set 实现

有三个通用的 Set 实现：HashSet，TreeSet 和 LinkedHashSet。HashSet 比 TreeSet 快，但是不提供顺序保证。如果需要使用 SortedSet 接口中的操作，或者如果要求按值的顺序迭代，使用 TreeSet。否则，使用 HashSet。大多数情况下会使用 HashSet。

LinkedHashSet 在某种意义上位于 HashSet 和 TreeSet 中间。作为带有链表的哈希表，它提供插入顺序的迭代，运行速度接近于 HashSet。LinkedHashSet 实现避免了 HashSet 提供的混乱顺序，同时又没有像 TreeSet 那样增加高昂的成本。

值得谨记在心的一件事是 HashSet 在条目数量和容量数量上呈线性迭代。因此初始容量选择太高会浪费空间和时间。另一方面，选择一个过低的容量会在强制增加其容量时浪费复制数据结构的时间。如果不指定一个初始的容量，默认是 16。在过去，选择一个素数作为初始容量有一些好处，但是现在不是这样了。HashSet 的容量总是以 2 的几何级数增长。使用 int 构造器指定初始的容量。下面这行代码给一个 HashSet 分配初始容量为 64：

```
Set<String> s = new HashSet<String>(64);
```

HashSet 类有另一个调整的参数，称为“加载因子”。否则，仅仅接受默认值；默认值总是最好的选择。LinkedHashSet 有同 HashSet 一样的调整因子，但是迭代时间不受容量的影响。TreeSet 没有调整因子。

2．特殊目的的 Set 实现

有两个特殊目的的 Set 实现：EnumSet 和 CopyOnWriteArraySet。EnumSet 是用于枚举类型的一个高性能的 Set 实现。一个枚举 set 集合中的所有成员必须是相同的枚举类型。内在地，它是由一个位向量代表的，典型地一个单独的 long 型。EnumSet 支持在枚举类型上的范围迭代。例如，给定一个用于一周的天数的枚举声明，可以在周一至周五之前迭代。EnumSet 类提供一个静态的工厂方法：

```
for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
      System.out.println(d);
```

EnumSet 还提供一个丰富的、类型安全的对传统的位标志的替代：

```
EnumSet.of(Style.BOLD,Style.ITALIC)
```

CopyOnWriteArraySet 是一个由写拷贝数组支持的 Set 实现。所有的可变操作，如 add()，set()和 remove()，通过对数组的一个新的拷贝来实现；这个实现只适用于很少修改但经常迭代的 Set 集合。它很适合于维护事件处理列表。

12.10.3 List 接口的实现

对 List 接口的实现也分为两类：通用目的的实现和特殊目的的实现。

1．通用目的的 List 实现

有两个通用目的的 List 实现：ArrayList 和 LinkedList。大多数时候，程序员或许都使用 ArrayList。它不必为 List 中的每一个元素分配节点对象，并且当它要在同一时刻移动多个元素时，它可以利用 System.arraycopy()。将 ArrayList 当做不带同步的向量 Vector。

如果频繁地添加元素到 List 的开始位置或迭代 List 以从 List 的内部删除元素，应该考虑使用 LinkedList。实际应用中，如果想使用 LinkedList，在做出决定之前，先用 LinkedList 和 ArrayList 都做一下应用程序的性能测试；ArrayList 通常会快一点。

ArrayList 有一个调整参数——初始容量，它代表 ArrayList 在增长之前可以持有的元素的数量。LinkedList 没有调整参数，有 7 个可选的操作，其中一个是 clone()。其他 6 个是 addFirst()、getFirst()、removeFirst()、addLast()、getLast()和 removeLast()。LinkedList 还实现了 Queue 接口。

2．特殊目的的 List 实现

CopyOnWriteArrayList 是一个由写拷贝数组支持的 List 实现。其本质上与 CopyOnWriteArraySet 相似。甚至在迭代期间，同步也不是必需的，而且迭代保证永远不抛出 ConcurrentModificationException。这个实现很适合用于维护事件处理列表。在事件处理列表中里很少发生改变，而遍历很频繁并且很耗时。

12.10.4 Map 接口的实现

对 Map 接口的实现可分为三类：通用目的、特殊目的和并发实现。下面依次讲解这三类实现方法。

1．通用目的的 Map 实现

三个通用目的的 Map 实现分别是 HashMap、TreeMap 和 LinkedHashMap。

- 如果需要 SortedMap 操作，或者按键排序的集合视图迭代，使用 TreeMap；
- 如果想要最大的速度而不关心迭代的顺序，使用 HashMap；
- 如果想要接近 HashMap 性能和按插入顺序迭代，使用 LinkedHashMap。

在这方面，Map 的情形与 Set 类似。同样地，在“Set 实现”一节中的任何事情都可应用于 Map 实现。

LinkedHashMap 提供两个 LinkedHashSet 不具备的性能。当创建一个 LinkedHashMap 时，可以基于键的访问顺序来排序，而不是基于插入顺序。换句话说，仅查找与一个键关联的值引起该键到 Map 的结尾。另外，LinkedHashMap 提供 removeEldestEntry()方法，该方法可以被覆盖以强加一个策略，该策略用于当新的映射被添加到 Map 中时，自动地删除过时的映射。这使得实现一个自定义缓冲非常地容易。

例如，下面这段代码覆盖将允许 Map 增长到 100 个条目的大小，然后每次一个新的条目被加入时，Map 将删除最老的那个条目，维护 100 个条目的稳定状态。

```
private static final int MAX_ENTRIES = 100;
```



```
protected boolean removeEldestEntry(Map.Entry eldest) {
   return size() > MAX_ENTRIES;
}
```

2．特殊目的的 Map 实现

有三个特殊目的的 Map 实现：EnumMap，WeakHashMap 和 IdentityHashMap。

3．并发的 Map 实现

在 java.util.concurrent 包中，包含有 ConcurrentMap 接口，它扩展自 Map，带有 putIfAbsent()、remove()和 replace()方法，以及该接口的 ConcurrentHashMap 实现。

12.10.5　Queue 接口的实现

对 Queue 接口的实现也分为两类：通用目的的实现和并发实现。

1．通用目的的 Queue 实现

LinkedList 实现了 Queue 接口，提供先进先出（FIFO）的队列操作如 add()、poll()等。PriorityQueue 类是一个基于堆数据结构的优先级队列。这个队列根据在构造时指定的顺序对元素进行排序，这既可以是元素的自然顺序，也可以是由一个显式的 Comparator 强行指定的顺序。

队列的提取操作，如 poll()、remove()、peek()和 element()，访问队列头部的元素。“队列头部”是按指定顺序排列的最前面的元素。

PriorityQueue 和它的迭代器实现了 Collection 和 Iterator 接口的所有可选地方法。以方法 iterator()提供的迭代器不保证以任何特定的顺序遍历 PriorityQueue 的元素。如果需要顺序遍历，使用 Arrays.sort(pq.toArray())。

2．并发队列实现

在 java.util.concurrent 包中包含有一系列同步的 Queue 接口和类。BlockingQueue 接口继承自 Queue 接口，带有当获取一个元素时等待队列变为非空的操作和当存储一个元素时使得队列的空间可用的操作。这个接口被以下的类实现。

- LinkedBlockingQueue：一个可选的基于链接节点的受限制的 FIFO 阻塞队列。
- ArrayBlockingQueue：一个基于数组的受限制的 FIFO 阻塞队列。
- PriorityBlockingQueue：一个基于堆的不受限制的阻塞优先级队列。
- DelayQueue：一个基于堆的基于时间调度的队列。
- SynchronousQueue：一个使用 BlockingQueue 接口的简单集合机制。

12.11　算法

Java 集合框架中的算法，指的是如何处理集合对象，例如对集合中的元素进行排序、查找、查找最小值最大值等。Java 集合框架中的算法都是多态算法。本节讲解一个算法实现的类 Collections。Collections 类提供了许多静态的方法，如排序、乱序等，可以用来操纵集合。

12.11.1 Collections 类

多态算法是由 Java 平台提供的可重用功能的一部分。所有这些算法都来自于 Collections 类，都采用静态方法的形式，这些静态方法的第一个参数是这些操作要执行的集合。由 Java 平台提供的大多数的算法都操作 List 实例，但有些操作任意的集合实例。本节简单描述下列算法。

- 排序：Collections 类的 sort()方法。
- 乱序：Collections 类的 shuffle()方法。
- 常规数据处理：Collections 类的 reverse()、fill()、copy()、swap()、addAll()方法。
- 查找：Collections 类的 binarySearch()方法。
- 组成：Collections 类的 frequency()、disjoint()方法。
- 查找极值：Collections 类的 min()、max()方法。

12.11.2 排序

排序算法重新对一个 List 排序，这样其元素根据一个顺序关系以升序排列。它提供了两个形式的操作。最简单的一种形式是接收一个 List 作为参数，并根据其元素的自然顺序对其排序。排序操作使用一个优化的归并排序（merge sort）算法，快速而稳定。

- 快速：实验证明它与高度优化的快速排序一样快速。快速排序一般被认为归并排序快但是不稳定。
- 稳定：它不对相同的元素进行重新排序。这对于在不同的属性上重复地对同一个 list 进行排序时非常重要。

【例 12.8】编写程序，按字典顺序（字母顺序）打印输出其命令行参数的内容。

```
import java.util.*;
public class Sort {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);//将命令行参数字符串数组转换为列表
        Collections.sort(list);      //调用 Collections 类的静态方法对 list 排序
        System.out.println(list);    //输出排序以后 list 中的元素
    }
}
```

编译并运行此程序：

```
java Sort i walk the line
```

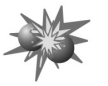
产生下面这样的输出结果：

```
[i,line,the,walk]
```

第二种形式的 sort 方法，除了一个 List 参数之外，还带有一个 Comparator 参数，并且使用 Comparator 对元素进行排序。

12.11.3 乱序

乱序算法所做的事与 Sort 所做的事相反，其作用是将 List 中已经存在的顺序打乱。这个算法基于从随机源的输入重新对 List 排序。这个算法在实现随机游戏时很有用。例如，



它可以被用于混排代表一副牌的扑克对象的 List。此外，它对生成测试用例很有用处。这个操作有两种形式：

- 一种带有一个 List 参数并使用一个默认的随机源。
- 另一个要求调用者提供一个 Random 对象用做随机源。

12.11.4　常规数据处理

Collections 类提供有 5 个算法用于在 List 对象上处理常规数据，所有这些算法都简洁明了：

- reverse()——反转一个 List 中元素的顺序。
- fill()——使用指定的值重写一个 List 中的每一个元素。这个操作对于重新初始化一个 List 是很有用的。
- copy()——带有两个参数，一个目标 List 和一个源 List，并将源中的元素复制到目标中，重新目标 List 的内容。目标 List 最少要与源 List 一样长。如果这更长的话，目标 List 中剩余的元素不受影响。
- swap()——交换一个 List 中指定位置的元素。
- addAll()——将所有指定的元素添加到一个 Collection 集合。

12.11.5　查找

二分查找算法在一个有序的 List 中查找指定的元素。这个算法有两种形式。

- 第一种带有一个 List 参数和一个要查找的元素参数。这种形式假设 List 是根据其元素的自然顺序按升序排过序的。
- 第二种形式除了 List 和要查找的元素外，还带有一个 Comparator，并假设 List 是根据指定的 Comparator 按升序排序的。可以先于调用 binarySearch()来调用排序算法。

两种形式的返回值都是一样的。如果 List 包含查找的元素，返回它的索引。如果不包含，返回值（-(插入位置)-1），插入位置是指被插入到 List 中的值的位置，或者比这个值大的第一个元素的索引，或者如果 List 中所有的元素都比指定的值小的话返回 list.size()。这保证如果且只有如果查找的元素被找到时返回值会大于 0。下面的代码片段中查找指定的元素并将其插入合适的位置，两种形式的 binarySearch()操作都可用。

```
int pos = Collections.binarySearch(list, key);
   if (pos < 0)
      l.add(-pos-1);
```

12.11.6　组成

频率和无关联算法测试组成一个或多个 Collection 的某些方面。

- frequency()：统计指定的元素在指定的集合中出现的次数。
- disjoint()：判断是否两个集合是无关联的。也就是说，它们是否没有包含相同的元素。

12.11.7　查找极值

min()和 max()算法返回在一个指定的 Collection 中包含的最小元素和最大元素。这两

个操作都有两种形式：

- 第一种简单的形式，是只带有一个集合参数并根据元素的自然顺序返回最小或最大元素。
- 第二种形式除了集合参数外还带有一个比较器 Comparator，并根据指定的 Comparator 返回最小或最大元素。



12.12 小结

本章主要学习了 Java 集合框架，包括接口、实现和算法。其中核心集合接口是 Java 集合框架的基础。实现是用于存储集合的数据对象，是实现了集合接口的类。

Java 的集合框架提供了很多数据结构的实现，熟练地掌握 Java 集合框架的使用，可以提高程序员的编码能力，优化程序结构。在学习本章时，重点了解 Java 集合框架的体系结构，掌握核心集合接口所定义的方法及其实现。特别是 List 接口的实现，要重点学习和掌握。Map 接口的实现和算法是本章的难点。

从下一章开始将向读者介绍如何使用 Java 语言创建图形界面应用程序。

3

第 3 篇

Java 图形图像篇

- 第 13 章　创建一个图形界面应用程序
- 第 14 章　使用 Swing 基本组件
- 第 15 章　使用 Swing 高级组件
- 第 16 章　程序界面布局
- 第 17 章　事件处理
- 第 18 章　图形、文本和图像处理

第13章 创建一个图形界面应用程序

到目前为止，已经学习了 Java 面向对象语言的基本语法和面向对象的概念。从本章开始，将应用这些面向对象的知识，学习如何创建带图形界面的 Java 应用程序。在 Java 中，要编写带图形界面的程序，需要学习并掌握 Swing 工具包的应用。



13.1 Java Swing 简介

Swing 工具包是 Java SE 平台的一部分，提供了一系列丰富的 GUI 组件，用来构建图形界面的应用程序。Swing 工具包包含丰富的组件集合，用于构建 GUI 和为 Java 应用程序添加交互性。Swing 包含从现代工具包所能获得的所有组件：表控件、列表控件、树控件、按钮和标签。

13.1.1 什么是 Swing

Swing 是 Java 平台提供的图形界面程序的开发工具包，包含很多与界面相关的类和接口，是 JFC 的重要组成部分。JFC 是 Java 基础类的简称，它包括一组用于构建图形用户接口（GUI）并添加丰富图形功能及交互性给 Java 应用程序的特性。Java 基础类的特性如表 13.1 所示。

表 13.1 Java 基础类的特性

特　性	描　述
Swing GUI 组件	从按钮到拆分窗格到表。许多组件具有排序、打印、拖放、命名等一些支持的特性
插件式外观支持	Swing 应用程序的外观是插件式的，允许选择程序外观。例如，同样的程序既可以应用 Java，也可以应用 Windows 的外观。另外，Java 平台支持 GTK+外观，它有数百个已经存在的可用的 Swing 程序外观。可以从各种来源获得更多可用的程序外观
易访问性 API	能使辅助技术如屏幕阅读器和盲文显示器等从用户接口获取信息
Java 2D API	能使开发者很容易地就将应用程序和 Applet 中的高质量的 2D 图形、文本和图像相结合。Java 2D 还包括扩展的 API，用于生成和发送高质量的输出到打印设备
国际化	允许开发者构建可以与全世界的用户以他们的语言和文化习惯进行交互的应用程序。开发者可以使用输入方法框架来构建能接受使用几千个不同字符的语言（如汉语、日语或朝鲜语）的文本的应用程序

本节主要关注 Swing 组件及其他的应用于 Swing 组件的 JFC 特性。Swing API 是强大的、灵活的。Swing API 一共有 18 个 public 类型的包，如表 13.2 所示。

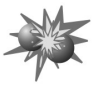


表 13.2　Swing API 中的包

javax.accessibility	javax.swing.plaf	javax.swing.text
javax.swing	javax.swing.plaf.basic	javax.swing.text.html
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html.parser
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.rtf
javax.swing.event	javax.swing.plaf.synth	javax.swing.text.tree
javax.swing.filechooser	javax.swing.table	javax.swing.text.undo

实际上，大多数程序只使用这些 API 中的一小部分。在本节中，大多数的代码只使用一到两个 Swing 的包：

- javax.swing
- javax.swing.event（并不总是需要）

13.1.2　Swing 特性

Swing 是 Java 基础类（Java Foundation Classes，JFC）的一部分。JFC 还包含对 GUI 程序很重要的其他特性，例如添加丰富的图形功能，以及创建可以用不同语言编写、可以由不同输入设备的用户使用的程序。

1．Swing GUI 组件

Swing 工具包包含丰富的组件：从基本的组件（例如按钮和复选框），到丰富复杂的组件（例如表和文本）。即使是看上去简单的组件（例如文本字段），也能提供复杂的功能（例如格式化的文本输入或者密码字段行为）。Swing 还包含有适合大多数需求的文件浏览器和对话框。如果 Swing 提供的组件不能完全满足要求的话，则可以利用基本的 Swing 组件功能创建自定义的组件。

2．Java 2D API

要使程序能可视化地传达信息，或者向 GUI 中添加图形、图像或动画，那么就需要使用 Java 2D API。因为 Swing 是构建在 2D 包之上的，所以很容易就能在 Swing 组件中使用 2D。使用 Java 2D 可以很容易地添加图像、绘制阴影等。

3．插件式程序外观支持（look and feel）

任何使用 Swing 组件的程序都可以选择程序外观。在 Swing 中，有一个 Synth 包，允许用户创建自己的程序外观。

Java 程序可以指定使用其要运行的平台的程序外观，或者可以指定总是使用 Java 的程序外观。如果不指定的话，由 UI 管理器负责选择合适的程序外观。

4．数据传递

在实际使用中，几乎所有的应用程序都可以通过剪切、复制、粘贴或拖放来传递数据。而 Swing 内置了对数据传递的支持，可以在一个应用程序的组件之间、Java 应用程序之间，以及 Java 程序和本地程序之间很好地进行数据传递工作。

5．国际化

这个特性，使开发者能构建与遍布全世界的用户以各自的语言和文化习惯进行交互的程序。可以创建接收使用数千个不同字符的语言的输入，如汉语、日语或朝鲜语。

Swing 提供的布局管理器，使进行界面设计时，更容易按 UI 要求的方向进行界面布局，在文本是自右向左的地区，UI 的出现也是自右向左的。这个支持是自动的：只需要对 UI 进行一次编码，然后 UI 就会自右向左或自左向右地工作，同样地，组件的大小也会随文本的位置而改变大小。

6．易访问性 API

残疾人在操作应用程序时，需要使用专门的辅助软件作为中介手段。这样的辅助软件需要获得正在运行的应用程序的大量信息，以便以可替代的媒介形式来表示，如使用屏幕阅读器合成语音读出屏幕内容，或者通过盲文显示表达内容；使用屏幕放大镜跟踪提示符和键盘焦点；屏幕上显示动态的键盘，用来控制菜单选项、工具栏项目和对话框控件；语音控制系统使用户可以通过声音控制操作。而 Java 的可访问性 API 使这些辅助软件能够获得所需的信息，并且通过程序操作构成图形用户界面的元素。

7．撤销框架 API

Swing 的 Undo Frameword（撤销框架）允许开发者对撤销和重做提供支持。撤销支持是内置在 Swing 的文本组件中的，例如文本框组件。对于其他组件，Swing 支持一个“无限”次数的撤销和重做动作。例如，能够很容易地撤销从一个表中添加和删除元素的操作。

8．灵活的部署支持

使用 Java 编写的程序，既可以将其创建为 Applet 小应用程序并使用 Java 插件运行在浏览器窗口中（Java 插件支持各种类型的浏览器，如 IE 浏览器、火狐浏览器（Firefox）和 Safari 浏览器），也可以使用 Java Web Start 创建一个能在浏览器中启动的应用程序。

由 Swing 提供的组件几乎都是轻量级组件。因为轻量级组件是绘制在包含它的容器中的，而不是绘制在自己的窗口中，所以轻量级组件最终必须包含在一个重量级容器中。

13.2 创建一个简单的图形界面程序

Java Swing 提供了如此丰富的 GUI 类库，以及强大的图形界面程序的开发功能，使得程序员可以轻松开发复杂的图形界面程序。本节通过使用 Swing 创建一个简单的图形界面程序，帮助读者快速了解 Java 桌面程序开发步骤，掌握简单的图形界面程序开发流程。

13.2.1 HelloWorldSwing.java 界面程序示例

图形界面程序，必须有一个用于显示的窗体。在 Swing 中，代表窗体的类是 javax.swing.JFrame，即位于 javax.swing 包中的 JFrame 类。程序中的其他组件必须包含在 JFrame 窗体上才能被显示在屏幕上，因此也称 JFrame 类为容器类，它能像一个容器一样容纳别的组件。

要使用 JFrame 类创建窗体，首先要做的就是在程序中导入 javax.swing.JFrame 类。另外，在本示例中还需要用到一个标签（也是一个 Swing 组件，在后面的章节中有详细的讲解），在 Swing 中，代表标签的组件类是 javax.swing.JLabel，因此也需要导入这个类。还用到 javax.swing.SwingUtilities，这是一个 Swing 工具类。导入这些类的代码如下所示。

```
import java.swing.JFrame;
import java.swing.JFrame;
import javax.swing.SwingUtilities;
```



下面创建一个最简单的窗体界面程序 HelloWorldSwing.java，它只有一个文件，显示一个窗体，在窗体中放置一个标签，在标签上显示“HelloWorld”。假定本章的程序都位于名为 chapter13 的包中，因此要在第一行代码处定义包。

【例 13.1】创建带有窗体界面的简单应用程序 HelloWorldSwing.java。

```
package chapter13;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
public class HelloWorldSwing {
    /* 创建一个 GUI 界面并显示. */
    private static void createAndShowGUI() {
        //创建并设置程序运行窗体.
        JFrame frame = new JFrame("HelloWorldSwing");   //创建带有标题的窗体
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                  //设置当关闭窗体时自动关闭窗口
        //添加"Hello World"标签.
        JLabel label = new JLabel("Hello World");//创建带有文字内容的标签对象
        frame.getContentPane().add(label);  //将获得的标签对象添加到内容面板中
        //显示窗体.
        frame.pack();                             //所有组件以首选大小显示
        frame.setVisible(true);                   //显示窗体
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作:
        //创建并显示本程序的 GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();               //创建窗体并显示
            }
        });
    }
}
```

注意保存此文件时，文件名的拼写及大小写必须与类名完全一致。类名为 HelloWorldSwing，因此文件名也必须取名为 HelloWorldSwing.java。

13.2.2　编译并运行 HelloWorldSwing 程序

源文件创建并正确保存以后，需要将其编译为二进制的字节码文件，才能被计算机识别并执行。使用下面的命令编译该程序：

```
javac -d . HelloWorldSwing.java
```

注意，这里的编译命令 javac 后跟了参数-d，用来指定将编译后的.class 放在此参数后面指定的目录中。这里为圆点(.)，代表当前目录。这时会自动在当前目录下创建文件夹 chapter13，以对应相应的包名，并将编译过后的 HelloWorldSwing.class 文件放在此文件夹下。成功编译以后，在当前目录下(与 chapter13 同一目录下)，使用 java 命令运行该程序：

```
java chapter13.HelloWorldSwing
```

或者

```
java chapter13/HelloWorldSwing
```

这两个命令的作用是一样的。程序启动，此时在屏幕上会出现窗体，窗体大小自动以最适合组件大小的状态出现。将光标放在窗体边框上，当光标变为左右向的箭头时，可以通过拖拉改变窗体的大小，如图 13.1 所示（在 Windows 系统下）。



图 13.1　HelloWorld 程序界面

在这里不对程序中的代码做过多说明。其中各组件类的用法会在下一章中详细地讲解。

13.3　使用 Swing 顶层容器

要理解上一节中程序代码的含义，需要了解一些 Swing 的背景知识。其中首先需要了解的是 Swing 顶层容器。所谓顶层容器，指的是容纳其他容器的容器组件，包括 JFrame 类、JWindow 类、JDialog 类和 JApplet 等。本节主要对常用的 JFrame 类和 JDialog 类进行介绍。

13.3.1　Swing 中的顶层容器类

Swing 提供三个顶层容器类：JFrame、JDialog 和 JApplet。当使用这些顶层容器类时，应该谨记以下原则：

- 要在屏幕上显示，每个 GUI 组件必须是一个“容器层级”的一部分。一个容器层级是一个拥有顶层容器作为根的组件树。
- 每个 GUI 组件只能属于一个容器。如果一个组件已经在一个容器当中，而又试图将它添加到另外一个容器中去，那么将会先从第一个容器中移除该组件，然后添加到第二个容器中去。
- 每一个顶层容器都有一个内容面板（content pane），一般而言，该内容面板包含（直接或间接地）顶层容器的 GUI 中的可视化组件，也就是说，各类可视化组件不直接放到顶层容器中，而是放在顶层容器的内容面板中。
- 可以选择添加一个菜单栏到顶层容器中。菜单栏按惯例位于顶层容器中，但是位于内容面板之外。在一些程序外观中，如 Mac OS 的程序外观，允许程序员决定将菜单栏放置到另外一个对于此程序外观来说更合适的地方，例如放到屏幕的上部。

需要注意的是，Swing 中还有一个名为 JInternalFrame 的内部窗体，它模仿 JFrame，但实际上内部窗体并不是真正的顶层容器。图 13.2 和 13.3 是由一个应用程序所创建的窗体的截图。这个窗体包含有一个绿色的菜单栏（没有菜单项）和窗体的内容面板，内容面板中有一个很大的空白的黄色标签。

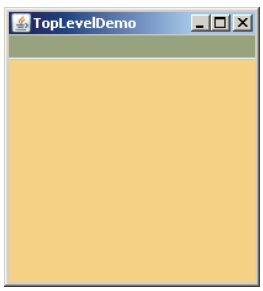


图 13.2　带有菜单栏和内容面板的窗体

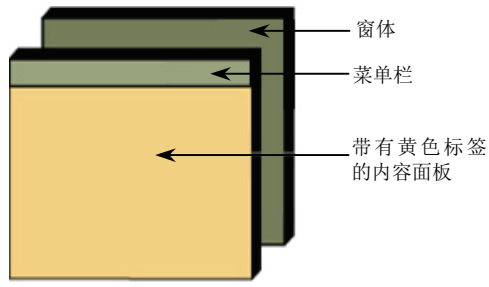


图 13.3　窗体、菜单和内容面板的关系图



【例 13.2】创建带有菜单栏和内容面板的窗体。

```
package chapter13;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/* TopLevelDemo.java 不需要其他文件. */
public class TopLevelDemo {
    // 创建一个 GUI 界面并显示. 出于线程安全的考虑，应该从事件分发线程调用此方法
    private static void createAndShowGUI() {
        //创建并设置窗体.
        JFrame frame = new JFrame("TopLevelDemo");  //创建带有标题的窗体对象
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                    //设置当关闭窗体时自动退出程序
        //创建一个菜单栏.将其背景设为绿色。
        JMenuBar greenMenuBar = new JMenuBar();
        greenMenuBar.setOpaque(true);               //将菜单栏背景设为不透明
        greenMenuBar.setBackground(new Color(154, 165, 127));
                                                    //设置菜单栏背景色
        greenMenuBar.setPreferredSize(new Dimension(200, 20));
                                                    //设置菜单栏的首选大小
        //创建一个黄色的标签并将其放入内容面板中
        JLabel yellowLabel = new JLabel();          //创建一个标签对象
        yellowLabel.setOpaque(true);                //将标签背景设为不透明的
        yellowLabel.setBackground(new Color(248, 213, 131));
                                                    //设置标签的背景颜色
        yellowLabel.setPreferredSize(new Dimension(200, 180));
                                                    //设置标签的首选大小
        //设置菜单栏并将标签添加到内容面板中.
        frame.setJMenuBar(greenMenuBar);            //将菜单栏添加到窗体上
        frame.getContentPane().add(yellowLabel, BorderLayout.CENTER);
                                                    //向窗体的内容面板添加标签
        //显示窗体.
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作:创建并显示本程序的 GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

编译此程序，如下所示。

```
javac -d . TopLevelDemo.java
```

注意，因为 TopLevelDemo.java 在包 chapter13 中，所以生成的 class 文件应该位于 chapter13 中，在 javac 命令后跟参数-d，再后面的圆点（.）代表在当前目录下创建包 chapter13，然后将编译后的 class 文件放于该包中。然后使用 java 命令运行此程序，如下所示。

```
java chapter13.TopLevelDemo
```

或

```
java chapter13/TopLevelDemo
```

虽然该示例程序在一个独立的应用程序中使用 JFrame，但是同样的原理也适用于 JApplet 和 JDialog。

13.3.2 顶层容器和容器层级

每一个使用Swing组件的程序都至少有一个顶层容器。这个顶层容器是容器层级的根，而容器层级包含所有出现在顶层容器中的 Swing 组件。图 13.4 是上面示例的图形用户接口（GUI）的容器层级。

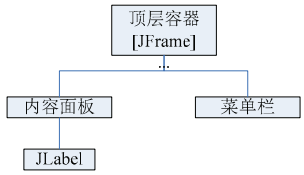


图 13.4 容器层级

作为一个原则，一个单独的、带有基于 Swing 的 GUI 的应用程序，至少要有一个带有 JFrame 作为其根的容器层级。例如，如果一个应用程序有一个主窗体和两个对话框，那么该应用程序有三个容器层级，因此有三个顶层容器。一个容器层级有一个 JFrame 作为它的根，其他两个容器层级每一个都有一个 JDialog 对象作为它的根（关于 JApplet，在本书中不做讨论）。

13.3.3 向内容面板中添加组件

上一节提到，作为一个 Swing 程序的原则，每一个顶层容器都有一个内容面板（content pane），各类可视化组件不直接放到顶层容器中，而是放在顶层容器的内容面板中。下面这行代码是在上一节的示例中，用来获得一个窗体的内容面板并向其添加黄色的标签的代码：

```
frame.getContentPane( ).add(yellowLabel, BorderLayout.CENTER);
```

如这行代码所示，通过调用 getContentPane()方法来找到顶层容器的内容面板。默认的内容面板是一个从 JComponect 继承过来的简单的中间容器，并使用 BorderLayout 作为其布局管理器。要使一个组件成为内容面板，使用顶层容器的 setContentPane()方法。例如：

```
//创建一个面板并向其中添加一个组件.
JPanel contentPane = new JPanel(new BorderLayout());  //创建一个 JPanel 实例对象
contentPane.setBorder(someBorder);                    //设置其边框
contentPane.add(someComponent, BorderLayout.CENTER);   //向面板中添加组件
contentPane.add(anotherComponent, BorderLayout.PAGE_END);
topLevelContainer.setContentPane(contentPane);      //将添加了组件的面板设
                                                      为容器的内容面板
```

其中 topLevelContainer 为一个顶层容器的名字。也可以这样编写代码，将一个组件添加到内容窗体的内容面板上：

```
frame.add(child);
```

child 组件将会被添加到内容面板中。这是因为，窗体的 add()方法及 remove()、setLayout()方法被重载了，以保证组件必须被添加到内容面板中。需要注意的是，只有这三个方法可以这样做。

13.3.4　给窗体添加一个菜单栏

从理论上说，所有的顶层容器都可以拥有一个菜单栏。然而在实际应用中，菜单栏通常只出现在窗体和 applet 中。要添加一个菜单栏到一个顶层容器，需要创建一个 JMenuBar 对象，它由菜单组成，然后调用 setJMenuBar()方法将其添加到容器中。在上面的 TopLevelDemo 示例中，使用下面的代码将一个菜单栏添加到其窗体中。

```
frame.setJMenuBar(greenMenuBar);
```

有关实现菜单和菜单栏的更多信息，见下一章“使用菜单”一节。

13.3.5　根面板

每一个顶层容器都依赖于一个隐含的中间容器，称为“根面板（root pane）”。根面板管理着内容面板和菜单栏，连同另两个容器。使用 Swing 组件的话程序员通常不需要了解根面板。然而，如果程序员需要截取鼠标操作或在多个组件上绘制，就应该认识根面板。图 13.5 是提供给一个窗体（以及每一个其他顶层容器）的一个根面板的组件列表。

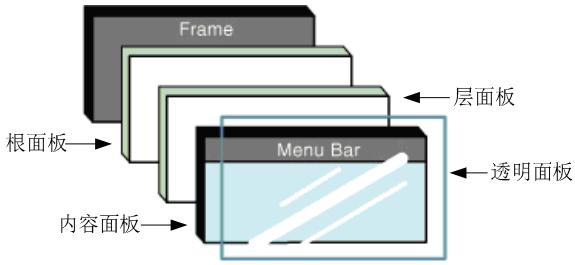


图 13.5　根面板组件列表

其中的内容面板和可选的菜单栏已经介绍过。内容面板添加的另两个组件是一个分层的面板（JLayeredPane 对象）和一个透明的面板（GlassPane 对象）。分层的面板包含菜单栏和内容面板，以及 Z 轴方向上的其他组件。透明面板经常被用于截取发生在顶层容器上的输入事件，还可以被用于在多个容器上绘制（更多的信息，见下一章“使用根面板”一节）。

13.4　使用 JFrame 类创建图形界面程序窗体

窗体是 JFrame 类的一个实例，是带有边框、标题并支持关闭和最小最大化按钮组件的一个窗口。带有 GUI 界面的应用程序一般最少要包含一个窗体。要创建一个依赖于另一个窗口的窗口（例如当另一个窗口被最小化时它也消失），要使用一个对话框来代替窗体。要创建一个在另一个窗口内出现的窗口，使用内部窗体。

13.4.1　创建并显示一个窗体

程序窗体是一个顶层窗口，带有标题和边框。窗体的大小包括边框的尺寸。边框的尺寸可以使用 getInsets 方法获得。因为边框被包括在窗体的总体大小内，所以边框隐藏了窗体的一部分，因而能渲染和显示子组件的窗体的可用区域为左上角顶点在（insets.left，insets.top）、大小为宽 width - (insets.left + insets.right)高 height – (insets.top + insets.bottom)。图 13.6 是由程序 FrameDemo.java 创建的一个极为简单的窗口的截图。



图 13.6　简单窗体

【例 13.3】创建并显示一个简单的应用程序窗体。

```
package chapter13;
import java.awt.*;
import javax.swing.*;
/* FrameDemo.java 不要求其他文件. */
public class FrameDemo {
    //创建一个 GUI 界面并显示. 出于线程安全的考虑，应该从事件分发线程调用此方法
    private static void createAndShowGUI() {
        //创建并设置程序运行窗口.
        JFrame frame = new JFrame("FrameDemo"); //创建带有标题的窗体对象
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                //设置当关闭窗体时退出程序
        JLabel emptyLabel = new JLabel("");     //创建不带文本内容的标签对象
        emptyLabel.setPreferredSize(new Dimension(175, 100));
                                                //设置标签的首选大小
        frame.getContentPane().add(emptyLabel, BorderLayout.CENTER);
                                                //将标签添加到内容面板中间
        //显示窗口
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作
        //创建并显示本程序的 GUI
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

使用如下的命令编译并运行此程序：

```
javac -d . FrameDemo.java
java chapter13.FrameDemo
```

这时，会看到如图 13.6 所示的窗口界面。在这个窗口中，只有一个无文本文字的标签组件。窗体有一个边框。在窗口的上方有标题和最小化、最大化及关闭按钮。

13.4.2 创建窗体的一般步骤

使用 Java 提供的 JFrame 类，创建一个应用程序的窗体非常简单，只需要生成一个 JFrame 类的对象，或者从 JFrame 类派生出一个新的类，稍做改动即可，这就是使用面向对象语言的好处。创建窗体一般分为以下几步。

① 创建窗体类 JFrame 的实例。

```
JFrame frame = new JFrame("FrameDemo");
```

这里使用了 JFrame 的一个带字符串参数的构造器，这个字符串参数用来设置窗体的标题，如图 13.6 所示。JFrame 还有一个不带参数的构造器，用来创建一个无标题的窗体，如下所示。

```
JFrame frame = new JFrame( );
```



② 设置窗体的标题内容（可选）。

```
frame.setTitle("FrameDemo");
```

③ 设置当关闭窗体的时候，会发生什么（可选）。

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

这个语句设置当关闭窗体时（单击窗口标题栏的关闭图标），退出程序。在只有一个窗体的情况下，这样做是很恰当的，关闭这个唯一的窗口会使得程序不可用。

④ 创建一些组件并将它们放置到窗体内。

```
//…创建空白标签 emptyLabel…
frame.getContentPane( ).add(emptyLabel, BorderLayout.CENTER);
```

这行代码将一个空白的标签添加到窗体的内容面板中。其中，getComtentPane()是 JFrame 的一个方法，用来获得当前窗体的内容面板，它返回一个 JContentPane 对象。对于具有菜单的窗体，在这里需要使用 JFrame 的 setJMenuBar()方法将菜单栏添加到窗体上。

⑤ 调整窗体大小。

```
frame.pack( );
```

pack 方法调用窗体的大小，使窗体的内容以最合适的大小显示。可替代 pack()方法的另外的方法是，通过显式地调用 setSize()或 setBounds()（setBounds()方法还设置窗体的位置）方法来建立一个窗体。一般来说，使用 pack()方法要优于调用 setSize()方法，因为 pack()方法让窗体的布局管理器来自主地决定窗体的大小，而布局管理器可基于所依赖的平台和其他影响组件大小的因素来做最合适的调整。

在上面的这个示例中，并不设置窗体的位置，不过可以很容易地通过方法 setLocationRelativeTo()或 setLocation()来设置。例如，下面的代码将一个窗体置于屏幕的中间：

```
frame.setLocationRelativeTo(null);
```

⑥ 显示窗体。

```
frame.setVisible(true);
```

调用 setVisible(true)方法使窗体显示在屏幕上。默认情况下，窗体是不显示的。在有的时候，可能会见到有使用 show()方法的，它和 setVisible()的方法是等价的，不过 show()已经过时，在新的 JDK 中，统一使用 setVisible(true)方法。

13.4.3　指定窗口装饰

窗口装饰，指的是窗口的外观样式。默认情况下，窗口装饰由本地窗口系统提供。不过，可以请求程序外观（look-and-feel）为窗体提供装饰。还可以指定窗体根本就没有窗口装饰（这是可用于窗体的一个特性），或者提供自己的装饰，或者使用全屏独占模式。

除了指定由谁来装饰窗口之外，还可以指定用哪个图标来代表这个窗口。如果窗口系统支持最小化，那么图标就被用于代表最小化窗口。大多数窗口系统或 look-and-feel 还在窗口装饰中显示图标。通常图标的大小为 16×16 像素。

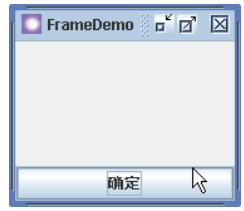
下面的截图 13.7～13.9 显示了三个相同的窗体，但是其窗口装饰不同。从每一个窗体中按钮的外观上可以看出，三个窗体都使用了默认的 Java 程序外观（跨平台的 look-and-feel）。



图 13.7 默认的窗口装饰



图 13.8 由 look-and-feel 提供的窗口装饰



13.9 带自定义图标的窗口装饰

其中第一个窗体（图 13.7）使用由窗口系统提供的装饰，也是默认的窗体装饰，从图 13.7 中可以看出，图中显示的是微软 Windows 系统的窗体装饰。第二个（图 13.8）和第三个窗体（图 13.9）使用由默认的 Java 程序外观（跨平台的 look-and-feel）提供的窗口装饰。其中第三个窗体（图 13.9）带自定义图标。如果希望上面的示例程序显示自定义图标并带有指定程序外观提供的窗口装饰，那么需要在程序中添加如下的代码：

```
//请求由程序外观提供的窗口装饰
JFrame.setDefaultLookAndFeelDecorated(true);    //这是 JFrame 的一个静态方法
//创建窗体
JFrame frame = new JFrame("新的窗口");
//设置窗体图标。图标的图像加载自一个文件
frame.setIconImage(new ImageIcon(imgURL).getImage());
```

注意这里代码的顺序，一定要先调用 JFrame 类的 setDefaultLookAndFeelDecorated()方法设置窗口装饰，然后再创建新的窗体，那么新的窗体才带有指定的窗口装饰。用 setDefaultLookAndFeelDecorated()设置的窗口装饰被用于后续所有创建的 JFrame。

如果想切换回窗口系统装饰，那么可以通过调用方法 JFrame.setDefaultLookAndFeel Decorated（false）来切换。有一些程序外观可能不支持窗口装饰，在这种情况下，使用窗口系统装饰。

【例 13.4】为上一节的程序添加相应的代码来设置窗口装饰和自定义图标。

```
package chapter13;
import java.awt.*;
import javax.swing.*;
/* FrameDemo.java 不要求其他文件. */
public class FrameDemo2 {
    //创建一个 GUI 界面并显示. 出于线程安全的考虑，应该从事件分发线程调用此方法
    private static void createAndShowGUI() {
        //设置窗口装饰
        JFrame.setDefaultLookAndFeelDecorated(true);
        //创建并设置程序运行窗口.
        JFrame frame = new JFrame("FrameDemo");
        //设置窗体图标。getFDImage()为自定义的方法，用来加载小图标
        frame.setIconImage(getFDImage( ));
        //设置当关闭窗口时，自动退出程序
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel emptyLabel = new JLabel("");      //创建一个不带文本的空标签
        emptyLabel.setPreferredSize(new Dimension(175, 100));
                                                 //设置标签的首选大小
        frame.getContentPane().add(emptyLabel, BorderLayout.CENTER);
                                                 //向内容面板添加标签
```



```
        //显示窗口
        frame.pack();
        frame.setVisible(true);
    }
    //加载图像
    protected static Image getFDImage() {
        java.net.URL imgURL = FrameDemo2.class.getResource("images/FD.jpg");
                                                          //获取图像文件路径
        if (imgURL != null) {
            return new ImageIcon(imgURL).getImage();      //返回创建的图像对象
        } else {
            return null;
        }
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作:创建并显示本程序的 GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

编译并运行此程序，程序运行结果如图 13.10 所示。

在这个程序中，设置窗体标题栏要显示的图标为 FD.jpg。要注意的是，在运行此程序之前，应该在存放 FrameDemo2.class 文件的 chapter13 文件夹内创建一个目录 images，然后将 FD.jpg 图像文件放在 images 目录下。

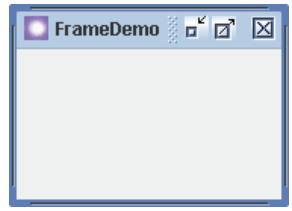


图 13.10　窗口装饰

13.4.4　响应窗口关闭事件

默认情况下，当用户关闭屏幕上的一个窗体时，窗体被隐藏。虽然不可见，但窗体仍然是存在的，并且程序可以再一次使其可见。如果想改变这种默认的行为，那么需要注册一个窗口监听器来处理窗口关闭事件，或者也可以使用 setDefaultCloseOperation()方法指定默认的关闭行为。甚至于两个方法都可以用。

setDefaultCloseOperation()方法的参数必须是以下常量值中的一个，其中前三个是在 WindowConstants 接口中定义的（JFrame、JInternalPane 和 JDialog 类实现了该接口）。

- DO_NOTHING_ON_CLOSE：当用户关闭窗口时，什么也不做。相反，程序应该使用一个窗口监听器来执行一些其他的 windowClosing()方法中的动作。
- HIDE_ON_CLOSE(默认情况下用于 JFrame 和 JDialog)：当用户关闭窗口时隐藏窗口。这会从屏幕上移除窗口，但是程序并没有结束，仍然具有显示的能力。
- DISPOSE_ON_CLOSE(默认情况下用于 JInternalPane)：当用户关闭窗口时隐藏并释放窗口。这会从屏幕上移除窗口并释放它所占用的资源。
- EXIT_ON_CLOSE(在 JFrame 类中定义的)：使用 System.exit(0)退出应用程序。推荐只在应用程序中使用此设置。如果用于小应用程序 applet 的话，会抛出一个 SecurityException 异常。

需要注意的是，当屏幕上只有一个窗口时，DISPOSE_ON_CLOSE 与 EXIT_ON_CLOSE 的结果是相同的，都是退出程序，释放资源。更准确地说，当 Java 虚拟机（JVM）中的最后一个可显示的窗口被释放时，虚拟机就会终止。

在任何窗口监听器处理窗口关闭事件以后，默认的关闭操作会被执行。例如，假定指定默认的关闭操作是 DISPOSE_ON_CLOSE，并实现一个窗口监听器，窗口监听器会判断本窗体是否是最后一个可视的窗体，如果是的话，保存一些数据并退出应用程序。在这些条件下，当用户关闭窗体时，窗口监听器将会首先被调用。如果它并不退出应用程序，那么接着执行默认的关闭操作：关闭并释放窗体。

13.4.5 窗体 API

下面的表 13.3～13.5 列出了经常使用的 JFrame 类的构造器和方法。其他还有一些方法是 JFrame 从 java.awt.Frame、java.awt.Window 和 java.awt.Component 类继承过来的。

因为每一个 JFrame 对象都有一个根面板，所以窗体支持在窗体子组件上的输入和绘图行为，能在不同的“层”上放置子组件，并支持 Swing 菜单栏。

用于窗体的 API 主要包括三类：

- 创建并设置一个窗体。
- 设置窗口的大小和位置。
- 与根面板相关的方法。

表 13.3 创建并设置窗体

方法或构造器	用　途
JFrame() JFrame(String)	创建一个初始不可见的窗体。String 参数为窗体提供一个标题。要使窗体可见，调用其 setVisible(true)方法
void setDefaultCloseOperation(int) int getDefaultCloseOperation()	设置或获取当用户按下窗体上的关闭按钮时的操作。可能的选择是： DO_NOTHING_ON_CLOSE HIDE_ON_CLOSE DISPOSE_ON_CLOSE EXIT_ON_CLOSE 前三个常量是在 WindowConstants 接口中定义而在 JFrame 中实现的。常量 EXIT_ON_CLOSE 是在 JFrame 类中定义的
void setIconImage(Image) Image getIconImage() (在 Frame 中定义的方法)	设置或获取代表窗体的图标。注意这里的参数是一个 java.awt.Image 对象，而不是一个 javax.swing.ImageIcon(或任何其他的 javax.swing.Icon 实现)
void setTitle(String) String getTitle() (在 Frame 中定义的方法)	设置或获取窗体的标题
void setUndecorated(boolean) boolean isUndecorated() (在 Frame 中定义的方法)	设置或获取是否此窗体应该被装饰。只有窗体还没有被显示时（还没有被 packed 或 shown）才有效。通常用于全屏独占模式，或使用自定义窗口
static void setDefaultLookAndFeelDecorated(boolean) static boolean isDefaultLookAndFeelDecorated()	决定后续创建的 Jframe 窗体应该拥有它们的由当前的程序外观所提供的窗口装饰（如边框、用于关闭窗口的小部件等）。注意这只是一个提示，有些程序外观可能不支持这个功能



表 13.4　设置窗体大小和位置

方　　法	用　　途
void pack() (在 Window 中定义的方法)	调整此窗口的大小，以适合其子组件的首选大小和布局
void setSize(int, int) void setSize(Dimension) Dimension getSize() (在 Component 中定义的方法)	设置或获得窗口的全部大小。其中整数参数指定相应的宽和高
void setBounds(int, int, int, int) void setBounds(Rectangle) Rectangle getBounds() (在 Component 中定义的方法)	设置或获得窗口的大小和位置。对于 setBounds 的带有整数参数的版本，前两个参数代表窗口左上角的 *x* 和 *y* 坐标，后两个整数参数代表窗口的宽和高
void setLocation(int, int) Point getLocation() (在 Component 中定义的方法)	设置或获得窗口左上角的位置。参数相应地代表 *x* 和 *y* 坐标
void setLocationRelativeTo(Component) (在 Window 中定义的方法)	定位窗口于指定组件的正中位置。如果参数是 null，那么窗口位于屏幕中央。应该在设置窗口大小以后再调用此方法

表 13.5　与根面板相关的方法

方　　法	用　　途
void setContentPane(Container) Container getContentPane()	设置或获得窗体的内容面板。内容面板包含窗体中的可视的 GUI 组件
JRootPane createRootPane() void setRootPane(JRootPane) JRootPane getRootPane()	创建、设置或获得窗体的根面板。根面板管理窗体的内部，包括内容面板、透明面板等
void setJMenuBar(JMenuBar) JMenuBar getJMenuBar()	设置或获得窗体菜单栏来管理一系列窗体的菜单
void setGlassPane(Component) Component getGlassPane()	设置或获得窗体的透明面板。可以使用透明面板来中途截取鼠标事件或在程序的 GUI 顶部进行绘图
void setLayeredPane(JLayeredPane) JLayeredPane getLayeredPane()	设置或获得窗体的层面板。可以使用窗体的层面板来将组件放置于其他组件的上面或后面



13.5　小结

本章学习如何使用 Swing 工具包创建带图形界面的 Java 应用程序。在学习本章时，重点理解顶层窗口的概念，特别是内容面板。因为 Java 应用程序中要求其他组件都要位于内容面板之上。在学习窗体及窗体事件响应时可能会有些理解上的困难，不用担心，只要了解其工作原理即可，在实际开发中，集成开发工具会给程序员完成很多基础性的工作的。

从下一章开始，将带领读者认识和使用 Swing 中的大多数基本组件。

第 14 章　使用 Swing 基本组件

要想创建漂亮而功能丰富的应用程序，首先要了解并掌握如何使用 Swing 包中提供的各种界面组件。本章将认识一个很重要的类 Jcomponent，学习并掌握 Swing 工具包中提供的基本控件和信息显示控件。



14.1　认识 JComponent 类

除了顶层容器以外，所有以“J”开头的 Swing 组件直接或间接地继承自 JComponent 类。例如，JPanel、JScrollPane、JButton 和 JTable 都继承自 JComponent。但是 JFrame 和 JDialog、JApplet 不是继承自 JComponent，因为它们是作为顶层容器实现的。所以了解 JComponent 类有助于理解大部分 Swing 组件所具有的方法和属性。

14.1.1　JComponent 特性

JComponent 类继承自 Container 类，而 Container 类又继承自 Component 类。Component 类包括从提供布局线索到支持绘制和事件的每一件事。Container 类支持添加组件到容器并对它们进行布局。本节的 API 列表总结了 Component 和 Container，也是 JComponent 最经常使用的方法。JComponent 类为其派生类提供如下的功能。

- 工具提示：通过使用 setToolTipText()方法指定一个字符串，可以为组件的使用者提供帮助。当鼠标停留在组件上时，会在组件附近的小窗口中显示指定的提示字符串。
- 绘制边框：使用 setBorder()方法，可以指定一个环绕组件边缘的边框（要绘制一个组件的内部，重载 paintComponent()方法）。
- 应用程序的插件式程序外观：在后台，每一个 JComponent 对象都有一个相对应的 ComponentUI 对象来执行所有的画图、事件处理、决定大小等用于 JComponent 的事务。准确地说，使用哪一个 ComponentUI 取决于当前的程序外观（look and feel），可以使用 UIManager.setLookAndFeel()方法来设置程序的外观。
- 自定义属性：可以将一个或多个属性（名称/对象对）与任何 JComponent 相关联。例如，布局管理器可能使用属性来将一个约束对象与它所管理的每一个 JComponent 组件联系起来。可以使用 putClientProperty()和 getClientProperty()方法



来读写属性。

- 支持布局：在 JComponent 类中增加了许多在 Component 类中没有的方法，如 setMinimumSize()、setMaximumSize()、setAlignmentX()、setAlignmentY()，用来设置组件的大小和排列方式。
- 支持易访问性：JComponent 类提供 API 和基本功能来帮助辅助技术如屏幕阅读器来从 Swing 组件获取信息。
- 支持拖放：JComponent 提供 API 来设置组件的传递处理，这是用于 Swing 拖放支持的基础。
- 双缓冲：可以平滑地进行屏幕绘制。
- 键盘绑定：当用户在键盘上按下一个键时，相应的组件可以做出响应。例如，当一个按钮获得焦点时，击空格键等同于在按钮上的一个鼠标单击。程序外观自动地在按下和释放空格键之间进行绑定。

14.1.2　Jcomponent API

JComponent 类从 Component 和 Container 继承了许多方法，并提供了许多新的方法。表 14.1～表 14.8 中总结了最经常使用的方法。

表 14.1　自定义组件外观

方　法	用　途
void setBorder(Border) Border getBorder()	设置或获取组件的边框
void setForeground(Color) setBackground(Color)	设置组件的前景色或背景色。前景色一般是用于在组件里绘制文本。背景色是组件的背景区域，假设组件是不透明的
Color getForeground() Color getBackground()	获得组件的前景色和背景色
void setOpaque(boolean) boolean isOpaque()	设置或获得组件是否是不透明的。一个不透明的组件使用背景色填充其背景
void setFont(Font) Font getFont()	设置或获得组件的字体。如果一个组件没有设置字体，返回它父组件的字体
void setCursor(Cursor) Cursor getCursor()	设置或获得组件及其包含的所有组件（除了子组件，它们有自己的光标设置）上显示的光标。例如：aPanel.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));

表 14.2　设置或获得组件状态

方　法	用　途
void setComponentPopup Menu (JpopupMenu)	为这个 JComponent 设置 JpopupMenu。UI 负责注册绑定和添加必需的监听器，这样在合适的时候 JPopupMenu 就会被显示。什么时候显示 JpopupMenu 依赖于程序的外观：一些可能显示在鼠标事件上，一些可能绑定于一个键上 如果 popup 是 null，并且 getInheritsPopupMenu 返回 true，那么 getComponentPoputMenu 将会被委托给其父类。这提供了一个途径，使所有的子类组件继承父类的弹出菜单
void　setTransferHandler (TransferHandler) TransferHandler getTransferHandler()	设置或删除 transferHandler 属性。TransferHandler 支持通过剪切、复制或粘贴到/从剪贴板和拖放操作来交换数据
void setToolTipText(String)	设置在一个工具提示中显示的文本

续表

方　法	用　途
Void setName (String) String getName()	设置或获得组件的名称。这在需要与一个不显示文本的组件相关联的文本时很有用
boolean isShowing()	决定组件是否显示在屏幕上。这意味着组件必须是可视的，并且它必须位于一个容器中，容器也是可视的并显示
Void setEnabled(oolean) oolean isEnabled()	设置或获得组件是否是可用的。一个可用的组件能对用户的输入作出响应并产生事件
Void setVisible(oolean) oolean isVisible ()	设置或获得组件是否是可视的。组件初始都是可视的，除了顶层容器之外

表 14.3　处理事件

方　法	用　途
void addHierarchyListener(hierarchyListener l) void removeHierarchyListener(hierarchyListener l)	添加或删除指定的层次结构监听器，当此容器所属的层次结构发生变化时能够接收此组件发出的层次结构变化事件。如果监听器 l 为 null，则不会抛出异常并且不执行操作
void addMouseListener(MouseListener) void removeMouseLIstener(MouseListener)	为此组件添加或从此组件删除指定的鼠标监听器。当用户使用鼠标与被监听的组件交互时，通知鼠标监听器。如果监听器为 null，则不会抛出异常并且不执行操作
void addMouseMotionListener(MouseMotion Listener) void removeMouseMotionListener(MouseMotion Listener)	为此组件添加或从此组件删除指定的鼠标移动监听器，当用户在被监听的组件的范围内移动鼠标时，通知鼠标移动监听器。如果侦听器 l 为 null，则不会抛出异常并且不执行操作
void addKeyListener(KeyListener) void removeKeyListener(KeyListener)	为此组件添加或从此组件删除指定的按键监听器。当用户在键盘上击键时和被监听的组件有键盘焦点时，通知按键监听器。如果监听器为 null，则不会抛出异常并且不执行操作
void addComponentListener(ComponentListener) void removeComponentListener(ComponentListener)	为此组件添加或从此组件删除指定的组件监听器。当被监听的组件被隐藏、显示、移动或改变大小时，通知组件监听器。如果监听器为 null，则不会抛出异常并且不执行操作
boolean contains(int, int) boolean contains(Point)	检查组件是否“包含”指定的点，其中 *x* 和 *y* 是相对于此组件的坐标系统定义的
Component getComponentAt(int, int) Component getComponentAt(Point)	返回包含指定点的组件或子组件。在组件重叠的地方，最上层的子组件被返回。所有包含 Component.contains()返回的点的组件中，与索引 0 最接近的组件被返回
Component setComponentZOrder(component comp, int index)	移动指定的组件到容器指定的 *z* 轴索引处 如果组件是其他容器的子组件，那么在添加到这个容器之前，先从原先的容器中删除。这个方法与 java.awt.Container.add(Component,int)方法的最大不同在于，当从原先的容器中删除该组件时，这个方法并不调用 removeNotify，除非有必要并且本地窗口系统允许。使用这种方式，如果组件具有键盘焦点，那么移到新的位置时仍然保持焦点 注意：*z* 轴索引决定了组件被绘制的顺序。在 *z* 轴较高索引的组件被首先绘制，而较低索引的组件被最后绘制。在组件重叠的地方，低 *z* 轴索引的组件在高 *z* 轴索引的组件之上绘制
Component getComponentZOrder(component comp)	返回容器内组件的 *z* 顺序索引。组件在 *z* 顺序层次结构中所处的位置越高，它的索引就越低。具有最低 *z* 顺序索引的组件最后一个被绘制，而它在层次结构中高于其他所有子组件



表 14.4　绘制组件

方　　法	用　　途
void repaint() void repaint(int, int, int, int)	要求组件的全部或部分被重绘。4 个 int 参数指定要被绘制的矩形的边界（*x*, *y*, 宽，高）
void repaint(Rectangle)	要求组件内指定的区域被重绘
void revalidate()	要求组件和它所影响到的容器被再次布局。通常不需要调用这个方法，除非在它显示之后显式地改变了一个组件的大小和位置，或者在它显示之后改变了层级结构。在 revalidate 之后，总是调用 repaint 方法
void paintComponent(Graphics)	绘制组件。重载此方法以实现绘制自定义组件

表 14.5　处理容器层级结构

方　　法	用　　途
Component add(Component) Component add(Component, int) void add(Component, Object)	添加指定的组件到本容器中。其中带一个参数的方法添加组件到容器的最后。int 参数说明在容器中新的组件的位置。Object 参数为当前的布局管理器提供布局约束
void remove(int) void remove(Component) void removeAll()	从本容器中删除一个或多个组件。int 参数说明要删除的组件在容器中的位置
JRootPane getRootPane()	获得包含本组件的根面板
Container getTopLevelAncestor()	获得组件的最顶层容器：Window、Applet 或 null（如果组件没有被添加到任何容器中时）
Container getParent()	获得组件的直接容器
int getComponentCount()	获得本容器中的组件数量
Component getComponent(int) Component[] getComponents()	获得本容器中一个或所有的组件。int 参数说明要获取的组件的位置
Component getComponentZOrder(int) Component[] getComponentZOrder()	返回容器中组件的 *z* 轴索引。组件在 *z* 轴上的层级越高，索引越低

表 14.6　布局组件

方　　法	用　　途
void setPreferredSize(Dimension) void setMaximumSize(Dimension) void setMinimumSize	设置组件的首选大小、最大大小和最小大小，以像素计算。首选大小说明组件的最佳大小。组件应该比最大大小小比最小大小大。要意识到，这只是提示，有可能被一些布局管理器所忽略
Dimension getPreferredSize() Dimension getMaximumSize() Dimension getMinimumSize()	获得组件的首选大小、最大大小和最小大小，以像素计算。许多 JComponent 类有 set 和 get 方法。对于这些非 JComponent 子类，不含相应的 set 方法，可以通过创建一个子类并覆盖这些方法来设置一个组件的首选、最大和最小大小
void setAlignmentX(float) void setAlignmentY(float)	设置水平和垂直对齐方式。这些值说明此组件将如何相对于其他组件对齐。参数值应该是介于 0～1 之间的小数。0 代表与原点对齐，1 代表与原点最远的地方对齐，0.5 是居中对齐，等等。这只是推荐提示，会被一定的布局管理器所忽略
float getAlignmentX() float getAlignmentY()	获得组件水平和垂直方向上的对方方式。对于非 JComponent 的子类，没有相应的 set 方法，通过创建一个子类并覆盖这些方法
void setLayout(LayoutManager) LayoutManager getLayout()	设置或获得组件的布局管理器。布局管理器负责容器内组件的大小和位置
void applyComponentOrientation(ComponentOrientation) void setComponentOrientation(ComponentOrientation)	设置此容器及容器中所有的组件的 ComponentOrientation 属性

表 14.7　获取大小和位置信息

方　法	用　途
int getWidth() int getHeight()	获得组件当前的宽度或高度，以像素为单位
Dimension getSize() Dimension getSize(Dimension)	获得组件当前的大小，以像素为单位。当使用带有一个参数的此方法时，调用者负责创建返回结果的 Dimension 实例
int getX() int getY()	获得组件原点相对于父组件的左上角的当前的 *x* 和 *y* 坐标，以像素为单位
Rectangle getBounds() Rectangle getBounds(Rectangle)	以 Rectangle 对象的形式获得组件的边界。边界指定此组件的宽度、高度和相对于其父级的原点位置。以像素为单位
Point getLocation() Point getLocation(Point)	获得组件相对于父组件的左上角的当前的位置
Point getLocationOnScreen()	返回相对于屏幕左上角的位置
Insets getInsets()	获得组件边框的大小

表 14.8　指定绝对大小和位置

方　法	用　途
void setLocation(int, int) void setLocation(Point)	设置组件相对于父组件左上角的位置，以像素为单位。带有两个 int 参数的方法，依次指定 *x* 和 *y*。当不使用布局管理器时，使用这些方法来放置组件
void setSize(int, int) void setSize(Dimension)	设置组件的大小，以像素为单位。带有两个 int 参数的方法依次指定宽和高。当不使用布局管理器时使用这些方法来设置组件的大小
void setBounds(int, int, int, int) void setBounds(Rectangle)	设置组件相对于父组件左上角的大小和位置，以像素为单位。带有 4 个 int 参数的方法，依次指定组件左上角 *x* 和 *y* 坐标，由 width 和 height 指定宽和高。当不使用布局管理器时，使用这些方法来放置组件和设置组件大小



14.2　基本控件

Swing 的基本控件主要是用于接收用户输入的简单组件。基本控件也可以用于显示简单的状态。基本控件包括：按钮组件 Jbutton、复选框组件 JCheckBox、下拉列表组件 JComboBox、列表框组件 JList、菜单组件 JMenu、单选按钮组件 JRadioButton、滑块组件 JSlider、微调控制组件 JSpinner、文本框组件 JTextFeld 和密码框组件 JPasswordField。

14.2.1　按钮组件 JButton

在 Swing 中，有很多种按钮，如普通按钮、复选框、单选按钮等，它们都是 AbstractButton 类的子孙类。所以要创建一个按钮，可以实例化这些类的对象。表 14.9 列出了 Swing 定义的 AbstractButton 类的子类。

表 14.9　AbstractButton 类的子类

类　名	描　述
JButton	普通按钮
JCheckBox	复选框按钮
JRadioButton	一组单选按钮中的一个
JMenuItem	菜单中的一个菜单项
JCheckBoxMenuItem	带有复选框的菜单项
JRadioButtonMenuItem	带有一个单选按钮的菜单项



续表

类　名	描　述
JToggleButton	实现由 JCheckBox 和 JRadioButton 所继承的 toggle 功能。可以被实例化或子类化以创建双状态按钮

在 AbstractButton 中定义了许多基本的按钮 API，这些 API 被所有的 Swing 按钮所共用。首先请看下面的代码，其中使用了在 AbstractButton 类中定义的最经常使用的 API。

【例 14.1】按钮 JButton 组件应用示例。

```
package chapter14;
//导入 javax.swing 包中的相关类
import javax.swing.AbstractButton;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.ImageIcon;
import java.awt.event.KeyEvent;
/*
 * ButtonDemo.java 要求有以下的文件：
 *   images/right.gif
 *   images/middle.gif
 *   images/left.gif
 */
public class ButtonDemo {
    protected JButton b1, b2, b3;       //创建三个按钮对象 b1,b2 和 b3
    JFrame frame;                       //声明窗体对象
    JPanel contentPane;                 //声明面板对象
    public ButtonDemo() {               //构造器
        //创建并设置窗口.
        frame = new JFrame("ButtonDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        contentPane = new JPanel();     //创建面板对象
        frame.setContentPane(contentPane);
                                //将面板对象 contentPane 设为窗体的内容面板
       //创建三个图片
        ImageIcon leftButtonIcon = createImageIcon("images/right.gif");
        ImageIcon middleButtonIcon = createImageIcon("images/middle.gif");
        ImageIcon rightButtonIcon = createImageIcon("images/left.gif");
        b1 = new JButton("使中间按钮不可用", leftButtonIcon);
                                //创建最左边显示的按钮对象
        b1.setVerticalTextPosition(AbstractButton.CENTER);
                                //设置按钮上文本相对于图标的垂直位置
        b1.setHorizontalTextPosition(AbstractButton.LEADING);
                                //设置按钮上文本相对于图标的水平位置
        b1.setMnemonic(KeyEvent.VK_D);      //设置按钮 b1 的键盘助记符为键 D
        b1.setActionCommand("disable");     //设置此按钮的动作命令
        b2 = new JButton("中间按钮", middleButtonIcon); //创建中间显示的按钮对象
        b2.setVerticalTextPosition(AbstractButton.BOTTOM);
                                //设置按钮上文本相对于图标的垂直位置
        b2.setHorizontalTextPosition(AbstractButton.CENTER);
                                //设置按钮上文本相对于图标的水平位置
        b2.setMnemonic(KeyEvent.VK_M);      //设置按钮 b2 的键盘助记符为键 D
```

```
        b3 = new JButton("使中间按钮可用", rightButtonIcon);
                                        //创建右边显示的按钮对象
        //使用默认的文本位置（居中）
        b3.setMnemonic(KeyEvent.VK_E);          //设置按钮 b3 的键盘助记符为键 D
        b3.setActionCommand("enable");          //设置此按钮的动作命令
        b3.setEnabled(false);                   //禁用此按钮
        // 设置按钮上的工具提示
        b1.setToolTipText("单击这个按钮使中间按钮不可用.");
        b2.setToolTipText("当单击时，这个中间按钮什么都不做.");
        b3.setToolTipText("单击这个按钮使中间按钮可用.");
    }
    /* 返回一个 ImageIcon，或者如果文件路径无效，返回 null. */
    protected static ImageIcon createImageIcon(String path) {
        java.net.URL imgURL = ButtonDemo.class.getResource(path);
                                                //获取要加载图像的路径
        if (imgURL != null) {
            return new ImageIcon(imgURL);       //返回加载的图像
        } else {                                //如果加载图像过程中出现异常
            System.err.println("找不到文件: " + path);     //输出相关信息
            return null;                        //返回 null
        }
    }
    // 创建 GUI 界面并显示
    private void createAndShowGUI() {
        //将三个按钮添加到窗体的内容面板中
        contentPane.add(b1);
        contentPane.add(b2);
        contentPane.add(b3);
        //显示窗口
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作:创建并显示本程序的 GUI
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new ButtonDemo().createAndShowGUI();
            }
        });
    }
}
```

使用下面的命令编译和运行此程序：

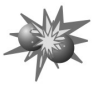
```
javac -d . ButtonDemo.java
java chapter14.ButtonDemo
```

运行结果如图 14.1 所示。



图 14.1　ButtonDemo.java 中的三个按钮

从图 14.1 可以看出，一个 Swing 按钮既可以显示文本，也可以显示图像，或两者同时显示。在 ButtonDemo.java 中，每一个按钮有它自己的文本并相对于图像位于不同的位置。将鼠标放在一个按钮上一秒钟，会出现一个方框文本提示。在代码中，对其中的方法都做了注释说明。



1．构造器

创建 JButton 按钮的构造器有 5 种，如表 14.10 所示。

表 14.10　JButton 类的构造器

构 造 器	描　　述
JButton()	创建不带有设置文本或图标的按钮
JButton(Action a)	创建一个按钮，其属性从所提供的 Action 中获取
JButton(Icon icon)	创建一个带图标的按钮
JButton(String text)	创建一个带文本的按钮
JButton(String text,Icon icon)	创建一个带初始文本和图标的按钮

2．设置按钮上文本与图标的相对位置

设置按钮上文本与图标的相对位置的代码为以下两行：

```
b1.setVerticalTextPosition(AbstractButton.CENTER);
                                        //设置按钮上文本相对于图标的垂直位置
b1.setHorizontalTextPosition(AbstractButton.LEADING);
                                        //设置按钮上文本相对于图标的水平位置
```

其中用于设置文本相对于图标的垂直位置的参数常量为下面三个值中之一：

- SwingConstants.CENTER（默认值，文本与图标都居中）。
- SwingConstants.TOP（文本在上，图标在下）。
- SwingConstants.BOTTOM（文本在下，图标在上）。

SwingConstants 为接口，AbstractButton 类实现了这一接口，所以在上面的代码中，通过 AbstractButton 类调用其从 SwingConstants 中继承过来的常量。

同样，用于设置文本相对于图标的水平位置的参数常量为下面 5 个值之一：

- SwingConstants.TRAILING（默认值）。
- SwingConstants.RIGHT（文本在右，图标在左）。
- SwingConstants.LEFT（文本在左，图标在右）。
- SwingConstants.CENTER（文本与图标居中对齐）。
- SwingConstants.LEADING（根据当地文字习惯，从左至右或从右至左）。

3．指定键盘助记符

在大多数程序外观中，用户可以通过按下 Alt 键和助记符来单击按钮。例如，上例中通过 Alt-D 将单击左边的按钮。其实现代码如下。

```
b1.setMnemonic(KeyEvent.VK_D);              //设置按钮 b1 的键盘助记符为键 D
```

4．设置工具提示

将鼠标放在一个按钮上一秒钟，会出现一个方框文本提示，显示相关的信息，一般是对该组件的说明或对该组件的操作说明。这用到了方法 setToolTipText。其代码如下。

```
b1.setToolTipText("单击这个按钮使中间按钮不可用.");
```

5．启用（或禁用）按钮

通过 AbstractButton 中定义的方法 setEnabled，可以设置按钮状态为可用或不可用。当不可用时，按钮变为灰色，这时不能响应任何操作。该方法参数为 boolean 型，当为 true

时，设置按钮可用；为 false 时，设置按钮不可用。例如上面的程序中的代码：

```
b3.setEnabled(false);
```

6．设置默认按钮

在一个顶层容器中，最多只能有一个按钮是默认按钮。默认按钮一般高亮显示，只要顶层容器拥有键盘焦点并且用户按下回车键时，相应的默认按钮就被单击。

通过在顶层容器的根面板上调用 setDefaultButton 方法来设置默认的按钮。下面是设置默认按钮的代码：

```
frame.getRootPane().setDefaultButton(b1);       //设置按钮 b1 为默认按钮
```

在 Windows 中，默认的按钮是拥有焦点的按钮，因此按下回车等价于单击有焦点的按钮。当没有按钮拥有焦点时，原始指定的作为默认按钮的按钮再次成为默认按钮。如果指定的默认按钮被禁用，那么默认按钮不起作用。

14.2.2 复选框组件 JCheckBox

JCheckBox 类提供了对复选框按钮的支持。还可以使用 JCheckBoxMenuItem 类将复选框放入菜单。因为 JCheckBox 和 JCheckBoxMenuItem 继承自 AbstractButton，所以 Swing 复选框具有所有一般按钮的特性。例如，可以指定带图像的复选框。

复选框与单选按钮相似，但是它们的选择模型不同。在一组复选框中可以选择任意数量的项：不选、选中一部分、选中所有的项。另一方面，一组单选按钮中只能选择一项。

【例 14.2】使用复选框的示例程序。

```
package chapter14;
import java.awt.*;
import javax.swing.*;
import java.awt.event.KeyEvent;
public class CheckBoxDemo {
    JCheckBox readingCheckBox;                          //声明复选框对象
    JCheckBox musicCheckBox;
    JCheckBox pingpongCheckBox;
    JLabel label;                                       //声明标签对象
    JFrame frame;                                       //声明窗体对象
    JPanel contentPane;                                 //声明内容面板
    public CheckBoxDemo() {                             //构造器
        frame = new JFrame("CheckBoxDemo");             //创建窗体对象
        //设置窗体在关闭时退出程序
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        contentPane = new JPanel();                     //创建面板对象
        frame.setContentPane(contentPane);              //设置内容面板
        label = new JLabel();                           //创建标签对象
        label.setText("爱好：");                         //设置标签上的文本
        readingCheckBox = new JCheckBox("读书");         //创建复选框.
        readingCheckBox.setMnemonic(KeyEvent.VK_C); //设置复选框的快捷键
        readingCheckBox.setSelected(true);        //将复选框默认设为选中状态
        musicCheckBox = new JCheckBox("听音乐"); //创建复选框
        musicCheckBox.setMnemonic(KeyEvent.VK_G);    //设置复选框的快捷键
        musicCheckBox.setSelected(true);          //将复选框默认设为选中状态
```



```
        pingpongCheckBox = new JCheckBox("乒乓球");  //创建复选框
        pingpongCheckBox.setMnemonic(KeyEvent.VK_H);//设置复选框的快捷键
        pingpongCheckBox.setSelected(true);      //将复选框默认设为选中状态
    }
    // 创建 GUI 界面并显示.
    public void createAndShowGUI() {
        //添加组件.
        contentPane.add(label);                  //向内容面板中添加标签
        contentPane.add(readingCheckBox);        //向内容面板中添加复选框
        contentPane.add(musicCheckBox);
        contentPane.add(pingpongCheckBox);
        //显示窗口.
        frame.pack();                            //设置组件以首选大小显示
        frame.setVisible(true);                  //将窗体设为可见
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作:创建并显示本程序的 GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              new CheckBoxDemo().createAndShowGUI();
            }
        });
    }
}
```

使用下面的命令编译和运行此程序：

```
javac -d . CheckBoxDemo.java
java chapter14. CheckBoxDemo
```

运行结果如图 14.2 所示。



图 14.2　CheckBoxDemo.java 中的复选框

通过 setText(String text)方法可以设置复选框的标签文本，通过 setSelected(Boolean b)方法可以设置复选框是否处于被选中状态。默认情况下未被选中，当设为 true 时，表示复选框被选中。

14.2.3　单选按钮组件 JRadioButton

单选按钮是一组在同一时刻只有一个按钮可以被选中的按钮。Swing 中支持单选按钮的是 JRadioButton 和 ButtonGroup 类。因为 JRadioButton 继承自 AbstractButton，所以单选按钮具有所有按钮的特性。例如，可以指定显示在单选按钮中的图标。

【例 14.3】使用单选按钮的示例程序：

```
package chapter14;
import java.awt.*;
import javax.swing.*;
import java.awt.event.KeyEvent;
public class RadioButtonDemo {
    JRadioButton manRadioButton,womanRadioButton;  //声明两个单选按钮对象
    JLabel label;                                  //声明标签对象
    JFrame frame;                                  //声明窗体对象
    JPanel contentPane;                            //声明面板对象
    public RadioButtonDemo () {                    //不带参数的构造器
```

```
        frame = new JFrame("RadioButtonDemo ");      //创建窗体对象
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                     //设置窗体关闭时自动退出程序
        contentPane = new JPanel();                  //创建面板对象
        frame.setContentPane(contentPane);           //设置内容面板
        label = new JLabel();                        //创建标签对象
        label.setText("性别: ");                     //设置标签上的文本
        manRadioButton = new JRadioButton("男");//创建单选按钮
        manRadioButton.setMnemonic(KeyEvent.VK_M);  //设置快捷键
        manRadioButton.setSelected(true);    //设置此单选按钮默认为选中状态
        womanRadioButton = new JRadioButton ("女"); //创建单选按钮
        womanRadioButton.setMnemonic(KeyEvent.VK_W);//设置快捷键
        //放入单选按钮组
        ButtonGroup group = new ButtonGroup();       //创建单选按钮组对象
        group.add(manRadioButton);       //将 manRadioButton 放入单选按钮组
        group.add(womanRadioButton);     //将 womanRadioButton 放入单选按钮组
    }
    // 创建 GUI 界面并显示
    public void createAndShowGUI() {
        //添加组件
        contentPane.add(label);                  //向内容面板中添加标签
        contentPane.add(manRadioButton);         //向内容面板中添加单选按钮
        contentPane.add(womanRadioButton);       //向内容面板中添加单选按钮
        //显示窗口
        frame.pack();                            //设置组件以首选大小显示
        frame.setVisible(true);                      //显示窗体
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作:创建并显示本程序的 GUI
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              new RadioButtonDemo ().createAndShowGUI();
            }
        });
    }
}
```



图 14.3　单选按钮

使用下面的命令编译和运行此程序：

```
javac -d . RadioButtonDemo.java
java chapter14. RadioButtonDemo
```

运行结果如图 14.3 所示。

试着分别单击两个单选按钮，会发现当选中一个按钮时，另一个按钮变为未选中状态。注意一定要将一组按钮放入一个单选按钮组 ButtonGroup 中，才能使选择唯一。

14.2.4　文本框组件 JTextField

Swing 中支持文本框组件的是 JTextField 类，用来接收用户输入的单行文本信息。如果需要为文本框设置默认文本，可以通过构造器 JTextField(String text)创建文本框对象。例如：

```
JTextField textField = new JTextField("请输入姓名");
```

也可以通过方法 setText(String str)为文本框设置文本信息，通过方法 getText()获取文本



框的信息。例如：

```
JTextField textField = new JTextField( );
textField.setText("请输入姓名");
System.out.pritnln(textField.getText());
```

【例 14.4】使用文本框的示例程序。

```
package chapter14;
import java.awt.*;
import javax.swing.*;
import java.awt.event.KeyEvent;
public class TextFieldDemo {
    JTextField nameTextField,addressTextField;  //声明两个文本框对象
    JFrame frame;                               //声明窗体对象
    JPanel contentPane;                         //声明面板对象
    JLabel nameLabel,addressLabel;              //声明两个标签对象
    public TextFieldDemo () {                   //不带参数的构造器
        frame = new JFrame("TextFieldDemo ");   //创建窗体对象
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                //设置当关闭窗体时自动退出程序
        contentPane = new JPanel();             //创建面板对象
        frame.setContentPane(contentPane);      //设置内容面板
        nameLabel = new JLabel("姓名：");       //创建代表姓名的标签对象
        addressLabel = new JLabel("地址：");    //创建代表地址的标签对象
        //创建文本框
        nameTextField = new JTextField ( );     //创建文本框对象
        nameTextField.setText("请输入姓名");    //设置文本框中默认显示的文本
        nameTextField.setHorizontalAlignment(JTextField.RIGHT);
                                                //将文本框中的文本设为靠右对齐
        nameTextField.setBounds(62,7,60,21);
                                          //设置文本框的大小（左上角坐标及长和宽）
        addressTextField = new JTextField ( );   //创建文本框对象
        addressTextField.setText("请输入地址");   //设置文本框中默认显示的文本
        addressTextField.setHorizontalAlignment(JTextField.CENTER);
                                                  //将文本框中的文本设为右对齐
        addressTextField.setBounds(62,35,120,21);//设置文本框的大小
    }
    // 创建 GUI 界面并显示
    public void createAndShowGUI() {
        //添加组件
        contentPane.add(nameLabel);             //向内容面板中添加姓名标签
        contentPane.add(nameTextField);         //向内容面板中添加姓名文本框
        contentPane.add(addressLabel);          //向内容面板中添加地址标签
        contentPane.add(addressTextField);      //向内容面板中添加地址文本框
        //显示窗口
        frame.pack();                           //设置组件以首选大小显示
        frame.setVisible(true);                 //显示窗体
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作:创建并显示本程序的 GUI
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              new TextFieldDemo ().createAndShowGUI();
```

```
            }
        });
    }
}
```

使用下面的命令编译和运行此程序：

```
javac -d . TextFieldDemo.java
java chapter14. TextFieldDemo
```

运行结果如图 14.4 和 14.5 所示。



图 14.4　文本框初始状态



图 14.5　文本框中文字的排列

其中，方法 setHorizontalAlignment(int alignment)用来设置文本框内容的水平对齐方式，其参数是 JTextField 类中定义的静态常量，共有如下三个值。

- JTextField.CENTER：文本居中。
- JTextField.LEFT：文本靠左显示。
- JTextField.RIGHT：文本靠右显示。

14.2.5　密码框组件 JPasswordField

Swing 中支持密码框组件的是 JPasswordField 类，它是 JTextField 的子类，为密码输入提供特殊的文本字符。出于安全的原因，密码框不能显示出用户输入的字符，要显示与用户输入不同的字符，例如一个星号‘*’。作为另外一种安全措施，密码框将它的值存储为一个字符数组，而不是字符串。当用户结束输入密码并单击时，如按下回车键，密码框响应一个动作事件。例如：

```
JPasswordField passwordField = new JPasswordField (16);
```

其中构造器参数为一个整数，表明密码框的大小为 16 个字符列大小。默认情况下，密码框为输入的每一个字符显示一个圆点。如果想改变回显的字符，调用 setEchoChar()方法。例如：

```
JPasswordField passwordField = new JPasswordField (16);
passwordField.setEchoChar('@');
```

虽然 JPasswordField 类继承了 getText()方法，但是还是应该使用 getPassword()方法来获得用户输入的密码值。getText()方法不仅缺少安全性，而且在将来它可能会返回可见的字符串（例如，“******”）来代替用户输入的字符串。

为了更进一步增强安全性，一旦已经使用完了由 getPassword()方法返回的字符数组，那么应该将该数组的元素全部清零。例如：

```
……
char[ ] input = passwordField.getPassword() ; //获得用户输入的密码的字符数组
……
Arrays.fill(input, '0') ;                    //为了安全，使用填充方法将数组清零
passwordField.selectAll( ) ;
resetFocus( ) ;
……
```



如果要对两个密码进行验证，检测两个密码是否相同，使用以下的 isPasswordCorrect() 方法，将需要判断的密码字符数组作为参数。例如：

```
private static boolean isPasswordCorrect(char[] input) {
    boolean isCorrect = true;                    //声明布尔变量，起标志变量的作用
    char[] correctPassword = { 'b', 'u', 'g', 'a', 'b', 'o', 'o' };
                                                 //字符数组，代表原始的密码
    if (input.length != correctPassword.length) {
                                                 //如果输入密码的长度与原始密码不同
         isCorrect = false;                      //标志变量为 false
    } else {                                     //否则，如果两个密码长度相同
       isCorrect = Arrays.equals (input, correctPassword);
                                                 //对两个密码数组进行判断
    }
    //清空密码.
    Arrays.fill(correctPassword,'0');            //出于安全考虑，用完以后要清除密码
    return isCorrect;                            //返回标志变量
}
```

【例 14.5】创建并显示一个密码框的程序。

```
package chapter14;
import javax.swing.*;
public class PasswordDemo {
     JPasswordField passwordField;  //声明密码框对象
     JFrame frame;                  //声明窗体对象
     JPanel contentPane;            //声明面板对象
     JLabel label;                  //声明标签对象
     public PasswordDemo () {       //构造器，通常在构造器中创建 swing 组件对象
         frame = new JFrame("PasswordDemo ");//创建窗体对象
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                    //设置当关闭窗体时自动退出程序
         contentPane = new JPanel();          //创建面板对象
         frame.setContentPane(contentPane); //设置内容面板
         label = new JLabel("请输入密码：");   //创建标签对象
         //创建文本框
         passwordField = new JPasswordField ( 16);//创建密码框对象
         passwordField.setEchoChar('@');          //设置密码框的回显字符为'@'
     }
     //创建 GUI 界面并显示
     public void createAndShowGUI() {
         //添加组件
         contentPane.add(label);                  //向内容面板中添加标签
         contentPane.add(passwordField);          //向内容面板中添加密码框
         //显示窗口
         frame.pack();                            //设置组件以首选大小显示
         frame.setVisible(true);                  //显示窗体
     }
     public static void main(String[] args) {
         //为事件分发线程预订一个工作：创建并显示本程序的 GUI
         javax.swing.SwingUtilities.invokeLater(new Runnable() {
             public void run() {
               new PasswordDemo ().createAndShowGUI();
```

```
            }
        });
    }
}
```



图 14.6　带自定义回显字符的密码框

编译并运行此程序，结果如图 14.6 所示。

在上面的程序中，使用密码框对象的 setEchoChar()方法设置密码框的回显字符为‘@’，同时在构造密码框时初始化其长度为 16。

14.2.6　组合框组件 JComboBox

JComboBox 组件可以让用户在多个选择项中选择其中一个，它有两种形式：不可编辑的和可编辑的。默认的形式是不可编辑的组合框，它的特征是拥有一个按钮和一个选择值的下拉列表。第二种形式，称为可编辑的组合框，它的特征是拥有一个文本框紧挨着一个小的按钮，用户可以在文本框中输入一个值或单击按钮显示下拉列表。这两种形式的组合框分别如图 14.7 和图 14.8 所示。



图 14.7 不可编辑组合框

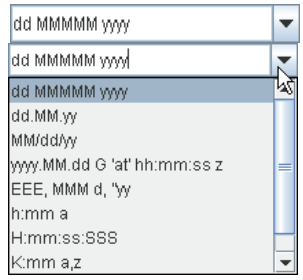


图 14.8　可编辑组合框

组合框节约屏幕空间，并且可编辑的组合框可以让用户快速地选择一个值，而不需要将选择限制在显示的值中。其他能够显示多选一的组件是单选按钮和列表。其中最直观的当然是单选按钮，但是组合框更适用于当对空间有限制或有超过一个的可用选择项时。当选择项目数很多时（如超过 20 个）或多项选择有效时，则使用列表更合适。

1．不可编辑组合框

下面的代码，创建一个不可编辑的组合框：

```
String[ ] petName = {"小狗","小花猫","鹦鹉","小兔子","小猪"};
JComboBox comboBox = new JComboBox(petName);
comboBox.setSelectedIndex(3);
```

还可以通过方法 addItem(Object item)和 insertItemAt(Object item,int index)向组合框中添加选项。例如：

```
comboBox.addItem("小鸭子");
comboBox.insertItemAt("小白鼠",4);
```

要获得用户在组合框中的选择项，可以使用下面两个方法：getItemAt(int)和 getSelectedItem()。这两个方法都返回一个 Object 对象。

要从组合框中删除选择项，可以使用下面三个方法：removeAllItems()、removeItemAt(int)和 removeItem(Object)。这三个方法都不需要有返回值。想知道组合框中总共有



多少个可供选择的项，使用方法 getItemCount()，它返回组合框中可供选择项的总数。

【例 14.6】使用组合框程序示例。

```
package chapter14;
import javax.swing.*;
public class ComboBoxDemo {
    JComboBox comboBox;                          //声明组合框对象
    JFrame frame;                                //声明窗体对象
    JPanel contentPane;                          //声明面板对象
    JLabel label;                                //声明标签对象
    public ComboBoxDemo () {                     //不带参数的构造器
        frame = new JFrame("ComboBoxDemo ");//创建窗体对象
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                 //设置当关闭窗体时，自动退出程序
        contentPane = new JPanel();              //创建面板对象
        frame.setContentPane(contentPane);       //设置内容面板
        label = new JLabel("请选择：");          //创建标签对象
        String[ ] petName = {"小狗","小花猫","鹦鹉","小兔子","小猪"};
        //创建不可编辑组合框
        comboBox = new JComboBox (petName);
        comboBox.addItem("小鸭子");              //为组合框添加内容项目
        comboBox.insertItemAt("小白鼠",4);       //向组合框中插入内容项
        comboBox.setSelectedIndex(3);            //设置组合框默认选中项为第 4 项
    }
    // 创建 GUI 界面并显示
    public void createAndShowGUI() {
        //添加组件
        contentPane.add(label);                  //向内容面板中添加标签
        contentPane.add(comboBox);               //向内容面板中添加组合框
        //显示窗口
        frame.pack();                            //设置窗体组件按首选大小显示
        frame.setVisible(true);                  //将窗体设置为可见（显示窗体）
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作:创建并显示本程序的 GUI
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              new ComboBoxDemo ().createAndShowGUI();
            }
        });
    }
}
```

编译并运行此程序，结果如图 14.9 所示。

在这个程序中，创建了一个组合框。默认情况下，该组合框是不可编辑的，读者可以尝试修改组合框中的内容。当然，也可以通过调用 setEditable(false)方法显式地将组合框设为不可编辑的。



图 14.9　不可编辑组合框

2．可编辑组合框

下面的代码，创建一个可编辑组合框：

```
String[] patternExamples = {
```

```
            "dd MMMMM yyyy",
            "dd.MM.yy",
            "MM/dd/yy",
            "yyyy.MM.dd G 'at' hh:mm:ss z",
            "EEE, MMM d, ''yy",
            "h:mm a",
            "H:mm:ss:SSS",
            "K:mm a,z",
            "yyyy.MMMMM.dd GGG hh:mm aaa"
    };
    . . .
    JComboBox patternList = new JComboBox(patternExamples);
    patternList.setEditable(true);
```

注意最后一行，通过方法 setEditable(true)方法将组合框设置为可编辑的。其他都简单，与不可编辑组合框类似。如果在不可编辑组件框示例程序中加上这个方法，那么创建的可编辑组合框如图 14.10 所示。



图 14.10　可编辑组合框

在上面的程序中，同样创建了一个组合框，但使用其方法 setEditable(true)，将其设为可编辑的。可以看出，在可编辑组合框中，可以输入下拉列表中没有的选项。

14.2.7　滑块组件 JSlider

Jslider 是一个让用户以图形方式在有界区间内通过移动滑块来选择值的组件。滑块可以显示主刻度标记和次刻度标记。如果空间有限，可选择使用 JSpinner 来代替。下面的代码显示如何创建一个滑块：

```
    static final int FPS_MIN = 0;       //代表最小值的常量
    static final int FPS_MAX = 30;      //代表最大值的常量
    static final int FPS_INIT = 15;     //代表初始滑块的值
    . . .
    JSlider framesPerSecond = new JSlider(JSlider.HORIZONTAL,FPS_MIN, FPS_MAX,
FPS_INIT);

    framesPerSecond.setMajorTickSpacing(10);    //设置主刻度标记的间隔，参数为整数
    framesPerSecond.setMinorTickSpacing(1);     //设置次刻度标记的间隔，参数为整数
    framesPerSecond.setPaintTicks(true);        //确定是否在滑块上绘制刻度标记
    framesPerSecond.setPaintLabels(true);       //确定是否在滑块上绘制标签
```

默认情况下，主刻度标记和次刻度标记的间隔都是 0。要想能看到刻度标记，必须显式地设置它们的间隔为一个非零的值，这通过调用 setPaintTicks(true)方法实现。除了显示刻度，还需要在主刻度的位置显示刻度值，那么在设置主刻度间隔后，调用 setPaintLabels(true)方法。另外，JDK1.6 中新添加了设置滑块字体的功能，允许程序员为 JSlider 组件设置一种字体。例如：

```
    Font font = new Font("Serif", Font.ITALIC, 15);
    framesPerSecond.setFont(font);
```

JSlider 组件还允许创建自定义的显示刻度值的标签。代码如下所示。

```
    //创建滑块
```



```
    JSlider framesPerSecond = new JSlider(JSlider.VERTICAL,FPS_MIN, FPS_MAX,
FPS_INIT);
    framesPerSecond.setMajorTickSpacing(10);      //设置主刻度标记的间隔为 10
    framesPerSecond.setPaintTicks(true);          //绘制刻度标记
    //创建显示刻度值的标签
    Hashtable labelTable = new Hashtable();
    labelTable.put( new Integer( 0 ), new JLabel("停止") );
    labelTable.put( new Integer( FPS_MAX/2 ), new JLabel("中速") );
    labelTable.put( new Integer( FPS_MAX ), new JLabel("快速") );
    framesPerSecond.setLabelTable( labelTable );  //设置刻度标签为自定义刻度标签
    framesPerSecond.setPaintLabels(true);
```

【例 14.7】编写程序，创建并显示一个滑块。

```
    package chapter14;
    import javax.swing.*;
    import java.awt.Font;
    import java.util.Hashtable;
    public class SliderDemo {
        static final int FPS_MIN = 0;                 //代表最小值的常量
        static final int FPS_MAX = 30;                //代表最大值的常量
        static final int FPS_INIT = 15;               //代表初始滑块的值
        JSlider framesPerSecond;                      //声明滑块对象
        JFrame frame;                                 //声明一个窗体对象
        JPanel contentPane;                           //声明一个面板对象
        JLabel label;                                 //声明一个标签对象
        public SliderDemo () {                        //不带参数的构造器
            frame = new JFrame("SliderDemo ");        //创建窗体对象
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                      //设置当关闭窗口时退出程序
            contentPane = new JPanel();               //创建面板对象
            frame.setContentPane(contentPane);        //设置内容面板
            label = new JLabel("请选择速度: ");       //创建标签对象
            //创建并设置滑块
            framesPerSecond = new JSlider(JSlider.VERTICAL,FPS_MIN, FPS_MAX,
FPS_INIT);
            framesPerSecond.setMajorTickSpacing(10); //设置主刻度标记的间隔为 10
            framesPerSecond.setMinorTickSpacing(1);  //设置次刻度标记的间隔为 1
            framesPerSecond.setPaintTicks(true);     //绘制主刻度标记
            //创建显示刻度值的标签
            Hashtable<Integer,JLabel> labelTable = new Hashtable<Integer,
JLabel>();
            labelTable.put( new Integer( 0 ), new JLabel("停止") );
            labelTable.put( new Integer( FPS_MAX/2 ), new JLabel("中速") );
            labelTable.put( new Integer( FPS_MAX ), new JLabel("快速") );
            framesPerSecond.setLabelTable( labelTable );
                                                  //设置刻度标签为自定义刻度标签
            framesPerSecond.setPaintLabels(true); //显示自定义刻度标签
            //设置刻度显示的字体
            Font font = new Font("Serif", Font.ITALIC, 15);   //创建字体
            framesPerSecond.setFont(font);        //设置滑块组件上的字体
        }
        // 创建 GUI 界面并显示
```

```
    public void createAndShowGUI() {
        //添加组件
        contentPane.add(label);              //向内容面板中添加标签
        contentPane.add(framesPerSecond);    //向内容面板中添加滑块
        //显示窗口
        frame.pack();                        //设置窗体中组件以首选大小显示
        frame.setVisible(true);              //显示窗体
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作:创建并显示本程序的 GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              new SliderDemo ().createAndShowGUI();
            }
        });
    }
}
```

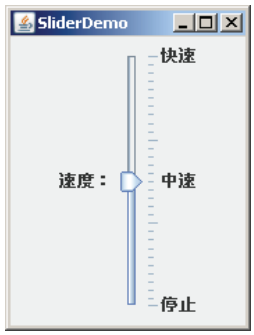


图 14.11　带自定义刻度标签的滑块

编译并运行此程序，结果如图 14.11 所示。

在上面的程序中，传递给 setLabelTable()方法的参数是一个 HashTable，在 HashTable 中的每一个键值对指定了一个标签的值和位置。哈希表的键必须是一个 Integer 类型，并且必须有一个在滑块限定范围内的值。哈希表中与每一个键相关联的值必须是一个 Component 对象。

还可以使用 JSlider 类的 createStandardLabels（int increment）方法创建一个将从滑块最小值处开始并以指定增量绘制文本标签的哈希表，使用 JSlider 类的 createStandardLabels（int increment,int start）方法创建一个将从起点处开始并以指定增量绘制文本标签的哈希表。还可以修改此方法返回的哈希表，以自定义哈希表。

从刚才的示例程序中可以看出，默认的滑块是垂直的。可以调用 JSlider 的 void setOrientation(int)方法设置其方向。参数值有两个：JSlider.HORIZONTAL（水平）和 JSlider.VERTICAL（垂直）。

14.2.8　微调控制组件 JSpinner

JSpinner 与组合框和列表相似，让用户从一个范围内选择一个值。与可编辑组合框一样，JSpinner 允许用户输入一个值，但是 JSpinner 没有下拉列表。因为 JSpinner 只显示当前值，不使用下拉列表显示可能的值，所以当选择项很多时，它经常被用于代替组合框或列表。需要注意的是，JSpinner 应该只被用于有明显顺序的值。

JSpinner 是一个复合组件，由三个小组件组成：两个小按钮和一个编辑器。编辑器可以是任何 JComponent，但是默认的是包含一个格式化文本框的面板。JSpinner 所有可能的值和当前的值由它的 model（MVC 中的模型）来管理。要创建一个微调选择器，首先要创建它的模型，然后将模型传递给 JSpinner 构造器。例如：

```
String[ ] monthStrings = {"元月",
                          "二月",
```



```
                        "三月",
                        "四月",
                        "五月",
                        "六月",
                        "七月",
                        "八月",
                        "九月",
                        "十月",
                        "十一月",
                        "十二月" };      //获得月份的名字
SpinnerListModel monthModel = new SpinnerListModel(monthStrings);
JSpinner spinner = new JSpinner(monthModel);
```

其中 SpinnerListModel 是一个模型，它的值由一个对象数组或一个列表对象来定义。还有一个模型是 SpinnerNumberModel，它支持有序的数字，可以是 double 对象、int 对象或 Number 对象。可以指定允许的最小值和最大值及步长（每一次增长或减少的量）。例如：

```
SpinnerModel model = new SpinnerNumberModel(currentYear,   //初始值
                                        currentYear-100,   //最小值
                                        currentYear+100,   //最大值
                                        1);                //步长
JSpinner spinner = new JSpinner(model);
```

要设置 JSpinner 当前显示的值，使用 model 的 setValue(Object)方法；要获得 JSpinner 当前选择的值，使用 model 的 getValue()方法。

【例 14.8】编写程序，创建并显示两个微调选择器。

```
package chapter14;
import javax.swing.*;
public class SpinnerDemo {
    String[ ] monthStrings = {"元月",
                        "二月",
                        "三月",
                        "四月",
                        "五月",
                        "六月",
                        "七月",
                        "八月",
                        "九月",
                        "十月",
                        "十一月",
                        "十二月" };                    //获得月份的名字
    static final int CURRENTYEAR = 2008;               //当前年份
    JSpinner monthSpinner,yearSpinner;                 //声明两个微调控制器组件对象
    SpinnerModel listModel,numberModel ;               //声明两个 SpinnerModel 对象
    JFrame frame;                                      //声明窗体对象
    JPanel contentPane;                                //声明内容面板
    JLabel monthLabel,yearLabel;                       //声明两个标签对象
    public SpinnerDemo () {                            //不带参数的构造器
        frame = new JFrame("SpinnerDemo ");            //创建窗体对象
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        contentPane = new JPanel();                    //创建面板对象
```

```
        frame.setContentPane(contentPane);      //设置内容面板
        yearLabel = new JLabel("请选择年份：");   //创建显示年份的标签
        monthLabel = new JLabel("请选择月份");      //创建显示月份的标签
        //微调选择器
        listModel = new SpinnerListModel(monthStrings);
        monthSpinner = new JSpinner(listModel);
        numberModel = new SpinnerNumberModel(CURRENTYEAR,  //初始值
                                             CURRENTYEAR -100, //最小值
                                             CURRENTYEAR +100, //最大值
                                             1);                  //步长
        yearSpinner = new JSpinner(numberModel );
       yearSpinner.setEditor(new JSpinner.NumberEditor(yearSpinner, "####"));
                                          //设置年份显示格式
    }
    // 创建 GUI 界面并显示
    public void createAndShowGUI() {
        //添加组件
        contentPane.add(yearLabel);    //向内容面板中添加显示年份的标签
        contentPane.add(yearSpinner); //向内容面板中添加显示年份的微调控制器
        contentPane.add(monthLabel);   //向内容面板中添加显示月份的标签
        contentPane.add(monthSpinner); //向内容面板中添加显示月份的微调控制器
        //显示窗口
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作:创建并显示本程序的 GUI
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              new SpinnerDemo ().createAndShowGUI();
            }
        });
    }
}
```

编译并运行此程序，结果如图 14.12 所示。



图 14.12　JSpinner 组件

JSpinner 组件是一个典型的 MVC 结构设计，其显示的数据，是由其模型对象 SpinnerListModel 来处理的。通过 SpinnerListModel 的带字符串数组参数的构造器，构造带有可选择数据的模型对象，然后再调用 JSpinner 组件的带有 SpinnerListModel 类型参数的构造器，构造 JSpinner 组件对象。

14.2.9　菜单组件 JMenu

菜单提供了另外一种节约空间的方式，让用户在多个选择项中选择一项。按惯例，菜单是唯一的，它们不与其他 UI 中的组件放在一起。菜单通常出现在菜单栏上，或者作为一个弹出菜单。一个菜单栏包含一个或多个菜单，一般出现在窗口的顶部。弹出菜单是一个不可见的菜单，直到用户在能弹出的组件上做出相应平台的鼠标动作，如单击右键，然后弹出菜单出现在鼠标下面。

图 14.13 显示了许多与菜单相关的组件：一个菜单栏，菜单，菜单项，单选按钮菜单，复选框菜单项，以及分隔符。正如图 14.13 所示，一个菜单项可以有图像或文本，也可以两者都有。另外还可以指定其他属性，如字体和颜色。

1．菜单组件的继承层次

图 14.14 显示了与菜单相关的类的继承层级。

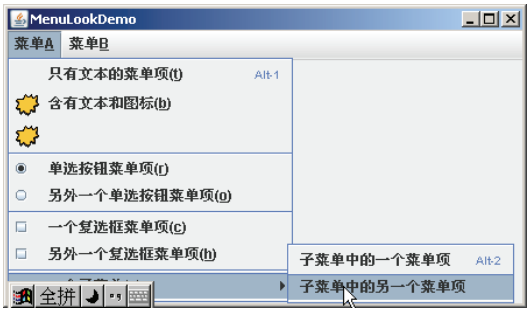


图 14.13　各种菜单

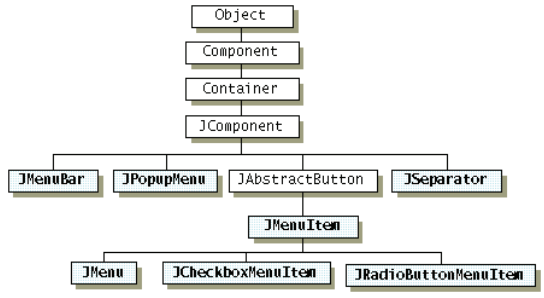


图 14.14　与菜单相关的类的继承层级

正如图 14.14 所示，菜单和菜单项都是简单的按钮。当一个菜单处理活动状态时，自动地弹出一个弹出菜单以显示菜单项。

2．创建菜单

菜单位于菜单栏之上，因此首先要创建一个菜单栏。在 Swing 中，用 JMenuBar 类代表菜单栏。下面是创建菜单栏的代码：

```
JMenuBar menuBar;
menuBar = new JMenuBar();
```

位于菜单栏上的是菜单。如图 14.13 所示的“菜单 A”和“菜单 B”。在 Swing 中，用 JMenu 类代表菜单。下面是创建菜单的代码：

```
JMenu menu, submenu;
menu = new JMenu("菜单 A");
menu.setMnemonic(KeyEvent.VK_A);    //设置快捷键
menu.getAccessibleContext().setAccessibleDescription("只有这个菜单有菜单项");
menuBar.add(menu);                  //将菜单添加到菜单栏
```

在菜单中含有可以响应用户操作的菜单项。在 Swing 中，用 JMenuItem 类代表普通菜单项，用 JRadioButtonMenuItem 类代表单选按钮菜单项，用 JCheckBoxMenuItem 代表复选框菜单项。下面是创建普通菜单项的代码：

```
JMenuItem menuItem;
menuItem = new JMenuItem("只有文本的菜单项(t)",KeyEvent.VK_T);
//menuItem.setMnemonic(KeyEvent.VK_T);      //也可以在构造器中取代这个方法
menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_1,
ActionEvent.ALT_MASK));//加速键设置
menuItem.getAccessibleContext().setAccessibleDescription("这个菜单项什么也
不做");
menu.add(menuItem);                         //将此菜单项加入到菜单中
//创建带有图标和文本的菜单项
ImageIcon icon = createImageIcon("images/middle.gif");
```

```
menuItem = new JMenuItem("含有文本和图标(b)", icon);
menuItem.setMnemonic(KeyEvent.VK_B);        //设置快捷键
menu.add(menuItem);
//也可以只有图标，没有文本显示
menuItem = new JMenuItem(icon);             //创建带图标的菜单项。参数为图标
menuItem.setMnemonic(KeyEvent.VK_D);        //设置菜单项的快捷键
menu.add(menuItem);                         //将菜单项添加到菜单栏中
```

下面是创建单选按钮菜单项的代码：

```
JRadioButtonMenuItem rbMenuItem;            //创建单选按钮对象
ButtonGroup group = new ButtonGroup();      //创建单选按钮组
rbMenuItem = new JRadioButtonMenuItem("单选按钮菜单项(r)");
rbMenuItem.setSelected(true);               //设置此单选按钮默认为选中状态
rbMenuItem.setMnemonic(KeyEvent.VK_R);
group.add(rbMenuItem);                      //将单选按钮菜单项添加到按钮组
menu.add(rbMenuItem);                       //将单选按钮菜单项添加到菜单栏中
```

下面是创建复选框菜单项的代码：

```
JCheckBoxMenuItem cbMenuItem;               //创建复选框形式菜单项
cbMenuItem = new JCheckBoxMenuItem("一个复选框菜单项(c)");
cbMenuItem.setMnemonic(KeyEvent.VK_C);
menu.add(cbMenuItem);
```

通常在菜单中，需要使用分隔线将不同功能的菜单项加以区分。使用下面的代码在菜单项之间添加分隔线：

```
menu.addSeparator();                        //添加分隔线
```

在菜单中除了可以包含菜单项之外，还可以包含子菜单（级联菜单）。在 Swing 中，子菜单也是菜单，也用 JMenu 类代表。下面是创建子菜单的代码：

```
submenu = new JMenu("A submenu");
submenu.setMnemonic(KeyEvent.VK_S);
menuItem = new JMenuItem("子菜单中的一个菜单项");
menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_2, ActionEvent.
ALT_MASK));
submenu.add(menuItem);      //向子菜单中添加菜单项
menu.add(submenu);          //将子菜单添加到菜单中
```

最后，设置窗体的菜单栏：

```
frame.setJMenuBar(menuBar);
```

【例 14.9】创建使用菜单的程序。

```
package chapter14;
import java.awt.*;
import java.awt.event.*;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.ButtonGroup;
import javax.swing.JMenuBar;
import javax.swing.KeyStroke;
import javax.swing.ImageIcon;
```



```
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.JFrame;
// MenuLookDemo.java 需要图标 images/middle.gif.
public class MenuLookDemo {
    JTextArea output;                                    //声明文本域
    JScrollPane scrollPane;                              //声明一个滚动面板
     /** 返回一个 JMenuBar(菜单栏)对象. */
    public JMenuBar createMenuBar() {
        JMenuBar menuBar;                                //声明菜单栏
        JMenu menu, submenu;                             //声明两个菜单
        JMenuItem menuItem;                              //声明一个菜单项
        JRadioButtonMenuItem rbMenuItem;                 //声明一个单选按钮菜单项
        JCheckBoxMenuItem cbMenuItem;                    //声明一个复选框菜单项
        //创建菜单栏
        menuBar = new JMenuBar();
        //构建第一个菜单
        menu = new JMenu("菜单 A");                      //创建菜单 A
        menu.setMnemonic(KeyEvent.VK_A);                 //设置快捷键
        menu.getAccessibleContext().setAccessibleDescription("只有这个菜单
有菜单项");
        menuBar.add(menu);                               //将菜单添加到菜单栏中
        //一组 JMenuItems
        menuItem = new JMenuItem("只有文本的菜单项(t)",KeyEvent.VK_T);
        //menuItem.setMnemonic(KeyEvent.VK_T);  //可以在构造器中取代这个方法
        menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_1,
ActionEvent.ALT_MASK));
        menuItem.getAccessibleContext().setAccessibleDescription("这个菜单
项什么也不做");
        menu.add(menuItem);                              //将菜单项添加到菜单中
        ImageIcon icon = createImageIcon("images/middle.gif");
                                                         //创建菜单项上显示的图像
        menuItem = new JMenuItem("含有文本和图标(b)", icon);
                                                         //创建带有图像的菜单项
        menuItem.setMnemonic(KeyEvent.VK_B);             //设置菜单项的快捷键
        menu.add(menuItem);                              //将菜单项添加到菜单中
        menuItem = new JMenuItem(icon);                  //创建只带图像的菜单项
        menuItem.setMnemonic(KeyEvent.VK_D);             //设置菜单项的快捷键
        menu.add(menuItem);                              //将菜单项添加到菜单中
        //一组单选按钮菜单项
        menu.addSeparator();                             //添加分隔符
        ButtonGroup group = new ButtonGroup();           //创建单选按钮组
        rbMenuItem = new JRadioButtonMenuItem("单选按钮菜单项(r)");
        rbMenuItem.setSelected(true);            //设置 rbMenuItem 为选中状态
        rbMenuItem.setMnemonic(KeyEvent.VK_R);   //设置快捷键
        group.add(rbMenuItem);           //将此单选按钮菜单添加到单选按钮组中
        menu.add(rbMenuItem);            //将此菜单项添加到菜单中
        rbMenuItem = new JRadioButtonMenuItem("另外一个单选按钮菜单项(o)");
        rbMenuItem.setMnemonic(KeyEvent.VK_O);
        group.add(rbMenuItem);
        menu.add(rbMenuItem);
```

```
            //一组复选框菜单项
            menu.addSeparator();                    //添加分隔线
            cbMenuItem = new JCheckBoxMenuItem("一个复选框菜单项(c)");
            cbMenuItem.setMnemonic(KeyEvent.VK_C);
            menu.add(cbMenuItem);
            cbMenuItem = new JCheckBoxMenuItem("另外一个复选框菜单项(h)");
            cbMenuItem.setMnemonic(KeyEvent.VK_H);
            menu.add(cbMenuItem);
            //一个子菜单
            menu.addSeparator();
            submenu = new JMenu("一个子菜单(s)");
            submenu.setMnemonic(KeyEvent.VK_S);      //设置子菜单的快捷键
            menuItem = new JMenuItem("子菜单中的一个菜单项");
            menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_2, Action
Event.ALT_MASK));
            submenu.add(menuItem);                   //将子菜单项添加到子菜单中
            menuItem = new JMenuItem("子菜单中的另一个菜单项");
            submenu.add(menuItem);                   //将子菜单项添加到子菜单中
            menu.add(submenu);                       //向菜单中添加子菜单
            //构建菜单栏中的第二个菜单
            menu = new JMenu("菜单 B");
            menu.setMnemonic(KeyEvent.VK_B);
            menu.getAccessibleContext().setAccessibleDescription("这个菜单什么
也不做");
            menuBar.add(menu);
            return menuBar;                          //返回构造好的菜单栏
        }
        /** 返回一个 ImageIcon 对象,如果路径无效,则返回 null. */
        protected static ImageIcon createImageIcon(String path) {
            java.net.URL imgURL = MenuLookDemo.class.getResource(path);
                                                     //获取图像文件的路径 URL
            if (imgURL != null) {
                return new ImageIcon(imgURL);
            } else {
                System.err.println("找不到文件: " + path);
                return null;
            }
        }
        // 创建 GUI 界面并显示
        private static void createAndShowGUI() {
            //创建和设置窗口
            JFrame frame = new JFrame("MenuLookDemo");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            //创建并设置菜单栏
            MenuLookDemo demo = new MenuLookDemo();
            frame.setJMenuBar(demo.createMenuBar());
            //显示窗口
            frame.setSize(450, 260);
            frame.setVisible(true);
        }
        public static void main(String[] args) {
            //为事件分发线程预订一个工作：创建并显示本程序的 GUI
            javax.swing.SwingUtilities.invokeLater(new Runnable() {
```



```
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

编译并运行此程序，结果如图 14.13 所示。正如代码中所示，为窗体设置菜单栏，使用 JFrame 类的 setJMenuBar 方法。要将一个菜单 JMenu 添加到一个菜单栏 JMenuBar，使用 JMenuBar 的 add(JMenu)方法。要将菜单项或子菜单添加到一个菜单，使用 JMenu 类的 add(JMenuItem)方法。



菜单项与其他组件一样，只能位于一个容器中。如果将第一个菜单中的菜单项添加到第二个菜单，那么该菜单项将会被从第一个菜单中移除，然后再添加到第二个菜单中。

14.2.10　列表框组件 JList

JList 组件有许多项供用户选择，这些选择项显示为一列或多列。通常将 JList 组件放在一个滚动面板中，当其中的选择项超过一定数量时，可以滚动查看。

1．创建一个 Model

JList 是将对数据的操纵交给一个模型（Model）处理。所以在使用列表之前，首先创建一个列表模型。有如下三种方式可以创建一个列表模型。

- DefaultListModel：默认情况下使用，也是大多数情况下使用的模型。
- AbstratcListModel：在这种方式下，程序员必须子类化 AbstractListModel 并实现其从 ListModel 接口继承的 getSize()和 getElementAt()方法。好处是可以由程序员自己来管理数据并调用相应的方法。
- ListModel：程序员必须实现所有的方法，管理所有的数据。

2．初始化列表

下面是创建并设置列表的代码：

```
JList list;
String[ ] address = {"河南省",
                     "河北省",
                     "湖南省",
                     "湖北省",
                     "广东省",
                     "广西壮族自治区",
                     "浙江省",
                     "山东省"};
list = new JList(address);                          //创建包含指定项的列表对象
list.setSelectionMode(ListSelectionModel.SINGLE_ SELECTION);
                                                    //设置列表项只能被单选
list.setLayoutOrientation(JList.HORIZONTAL_WRAP);  //设置组件的文字方向
list.setVisibleRowCount(-1);        //默认不选中任何一项
...
```

```
JScrollPane listScroller = new JScrollPane(list);
                                        //创建滚动面板，注意将 JList 对象作为参数
listScroller.setPreferredSize(new Dimension(250, 80));
                                        //设置滚动面板的首选大小
```

在上面的代码中，通过传递给列表构造器一个数组来初始化一个 JList 对象。另外，JList 还有其他的构造器，允许以一个 Vector 或实现了 ListModel 接口的 Model 来初始化列表。如果使用数组或 Vector 来初始化一个列表，构造器隐含地创建一个默认的列表 Model。默认的列表 Model 是不可变的，这意味着不能添加、删除或替换列表中的项。要创建一个列表项可以被个别地进行改变的列表，需要将列表的 Model 设置为一个可变列表 Model 类的实例，例如 DefaultListModel 的实例。

可以在创建列表时设置列表的 Model，也可以通过调用 setModel()方法来设置。调用方法 setSelectionMode()用来指明用户能选择多少项，以及所选择项是否必须是连续的。

对方法 setLayoutOrientation()的调用可以让列表在多个列上显示数据。此方法有三个常量参数，分别代表一种列表中数据项的排列方法。

- JList.HORIZONTAL_WRAP：指明列表项应该自左向右排列，直到一行的最后再换一个新行。
- JList.VERTICAL_WRAP：指明列表项应自上向下排列，直到一列排满为止，再换一个新列。
- JList.VERTICAL：列表项只排一列。

下面的图 14.15～图 14.17 分别显示了这几种列表项的排列方式。

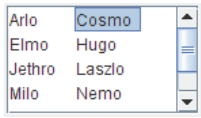


图 14.15　HORIZONTAL_WRAP

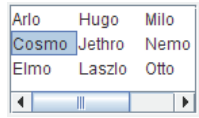


图 14.16　VERTICAL_WRAP

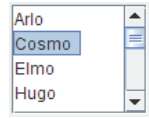


图 14.17　VERTICAL

与调用 setLayoutOrientation()方法相结合，调用 setVisibleRowCount(-1)使在屏幕可用空间中尽可能显示最大数量的列表项。如果 setVisibleRowCount()方法的参数为一个大于 0 的整数，那么用来设置不使用滚动条可以在列表中显示的首选行数。

3．在一个列表中选择列表项

JList 使用 ListSelectionModel 的实例来管理选择列表项的操作。默认情况下，列表选择模式允许同时选择多个连续或不连续的列表项。可以通过方法 setSelectionMode()来指定一个不同的选择模式。表 14.11 列举了三种列表项选择模式。

表 14.11　列表项选择模式

模　式	描　述	示　例
SINGLE_SELECTION	同时只能选择一个列表项。当用户选择一个列表项时，任何先前所选择的列表项会首先被取消选择	one two three four five
SINGLE_INTERVAL_SELECTION	连续多选。当用户重新选择一个新的范围时，原来的选择项会首先被取消选择	one two three four five

续表

模　式	描　述	示　例
MULTIPLE_INTERVAL_SELECTION	默认值。任何形式的列表项结合都可以被选择。用户必须显式地取消选择项	

对于 SINGLE_SELECTION 模式，要获得用户在列表的选择项，使用 getSelectedIndex()方法，返回一个整数，代表用户所选择的列表项的索引值。对于其他模式的多项选择，也有相应的方法用来获取用户的选择。具体请查阅 API 文档。

4．向列表中添加列表项和从列表中删除列表项

要向列表中添加列表项或从列表中删除列表项，需要使用可变的列表模型对象，并初始化模型对象，然后用这个列表模型来创建一个列表。如以下代码所示。

```
DefaultListModel listModel = new DefaultListModel();
                                              //创建 DefaultListModel 对象
listModel.addElement("河南省");               //向模型对象中添加元素
listModel.addElement("河北省");
listModel.addElement("湖南省”);
listModel.addElement("湖北省”);
JList list = new JList(listModel);
```

使用 addElement()会将新的列表项添加在列表的最后。要从列表中删除一个列表项，要先选择要删除的列表项，然后调用列表模型的 remove()方法删除。如以下代码所示。

```
int index = list.getSelectedIndex();
listModel.remove(index);
```

要向列表中插入一个列表项到指定的索引处，使用列表模型的 insertElementAt()方法，如以下代码所示。

```
listModel.insertElementAt("广东省",2);
```

【例 14.10】创建使用 JList 的程序。

```
package chapter14;
import javax.swing.*;
import java.awt.Dimension;
public class ListDemo {
     JList list;                                          //声明列表对象
     DefaultListModel listModel;                          //声明 ListModel 对象
     JFrame frame;                                        //声明窗体对象
     JPanel contentPane;                                  //声明面板对象
     JLabel label;                                        //声明标签对象
     JScrollPane listScroller;                            //声明滚动面板对象
     public ListDemo () {
         frame = new JFrame("ListDemo");                  //创建窗体对象
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         contentPane = new JPanel();                      //创建面板对象
         frame.setContentPane(contentPane);               //设置内容面板
         label = new JLabel("请选择省份：");              //创建标签对象
         //创建列表
         listModel = new DefaultListModel();              //创建 ListModel 对象
         listModel.addElement("河南省"); //向 listModel 对象中添加字符串对象
```

```
            listModel.addElement("河北省");
            listModel.addElement("湖南省");
            listModel.addElement("湖北省");
            list = new JList(listModel);    //从 listModel 构造 JList 实例
            list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
                                            //设置列表项只能被单选
            list.setLayoutOrientation(JList.VERTICAL);
                                            //设置排列方向为垂直方向
            list.setVisibleRowCount(-1);    //设置全部列表项可见
            list.setSelectedIndex(0);       //设置默认的选择项为第一个列表项
            //将列表添加到滚动面板
            listScroller = new JScrollPane(list);
                                            //创建滚动面板对象，并添加列表对象到其中
            listScroller.setPreferredSize(new Dimension(250, 80));
                                            //设置滚动面板的首选大小
        }
        // 创建 GUI 界面并显示
        public void createAndShowGUI() {
            //添加组件
            contentPane.add(label);
            contentPane.add(listScroller);
            //显示窗口
            frame.pack();
            frame.setVisible(true);
        }
        public static void main(String[] args) {
            //为事件分发线程预订一个工作：创建并显示本程序的 GUI
            javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                  new ListDemo ().createAndShowGUI();
                }
            });
        }
}
```

编译并运行上述程序，结果如图 14.18 所示。



图 14.18　JList 列表

JList 组件是一个典型的 MVC 结构设计，其显示的数据，是由其模型对象 DefaultListModel 来处理的。通过 DefaultListModel 的 addElement()方法，将要显示的对象添加到模型当中，然后再调用 JList 组件的带有 DefaultListModel 类型参数的构造器，构造 JList 组件对象。

14.3　不可编辑的信息显示控件

Swing 的不可编辑的信息显示控件只用于给用户提供信息，不能编辑。这些组件包括：标签组件 JLabel、进度条组件 JProgressBar、分隔线组件 JSeparator 和工具提示组件 JtoolTip。下面逐一对这些组件进行介绍。

14.3.1　标签组件 JLabel

在 Swing 中，最经常用来显示信息的就是标签 JLabel。使用 JLabel 类，可以显示不能



选择和修改的文本和图像。如果需要创建一个组件用来显示字符串或图像或两者都有，可以使用 JLabel 或 JLabel 的派生类。

通过在标签文本中指定 HTML 代码，可以赋予标签各种特性，如多行文本显示、多种字体显示或多种颜色。如果标签只使用单一的颜色或字体，使用 setForeground()或 setFont()来设置。默认情况下，标签是透明的。如果需要绘制标签的背景，使用下面的代码将其不透明属性设为“true”，如下所示。

```
label.setOpaque(true);
```

如果创建只包含文本的标签，将要显示的文本传递给 JLabel 的构造器。如以下代码所示。

```
label1 = new JLabel("只含有文本的标签");
```

如果创建只包含图标的标签，将要显示的图标传递给 JLabel 的构造器。如以下代码所示。

```
ImageIcon icon = createImageIcon("images/middle.gif");
label2 = new JLabel(icon);
```

方法 createImageIcon()与前面几节示例中的同名方法相同，用来获得一个图像。如果创建包含图标和文本的标签，使用如下代码。

```
ImageIcon icon = createImageIcon("images/middle.gif");
label3 = new JLabel("文本与图标",icon,JLabel.CENTER);
//设置文本相对于图标的位置
label3.setVerticalTextPosition(JLabel.BOTTOM);
label3.setHorizontalTextPosition(JLabel.CENTER);
```

经常，使用标签来描述另外一个组件。这时可以通过使用 setLabelFor()方法标识标签所描述的组件来提高程序的易访问性。例如：

```
pwdLabel.setLabelFor(pwdField);
```

【例 14.11】创建使用 JLabel 的示例程序。在这个程序中，创建并显示了三种类型的标签。

```
package chapter14;
import javax.swing.*;
import java.awt.GridLayout;
public class LabelDemo {
     JFrame frame;                                          //声明窗体对象
     JPanel contentPane;                                    //声明面板对象
     JLabel label1,label2,label3;                           //声明两个标签对象
     public LabelDemo () {                                  //不带参数的构造器
         frame = new JFrame("LabelDemo ");                  //创建窗体对象
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         contentPane = new JPanel(new GridLayout(3,1));//三行一列的面板
         frame.setContentPane(contentPane);                    //设置内容面板
         ImageIcon icon = createImageIcon("images/middle.gif");
                                                        //创建要在标签上显示的图标
         label1 = new JLabel("只含有文本的标签");  //创建只含有文本的标签对象
         label2 = new JLabel(icon);                     //创建只含有图标的标签对象
         label3 = new JLabel("文本与图标",icon,JLabel.CENTER);
                                                        //创建含有文本和图标的标签对象
         //设置文本相对于图标的位置
         label3.setVerticalTextPosition(JLabel.BOTTOM);
                                                        //设置文本在下面图标在上
         label3.setHorizontalTextPosition(JLabel.CENTER);
```

```
    }
    /**返回一个 ImageIcon 对象,如果文件路径无效，则返回 null. */
    public ImageIcon createImageIcon(String path) {
        java.net.URL imgURL = LabelDemo.class.getResource(path);
        if (imgURL != null) {
             return new ImageIcon(imgURL);
        } else {
             System.err.println("找不到文件: " + path);
             return null;
        }
    }
    // 创建 GUI 界面并显示
    public void createAndShowGUI() {
         //添加组件.
         contentPane.add(label3);
         contentPane.add(label2);
         contentPane.add(label1);
         //显示窗口
         frame.pack();
         frame.setVisible(true);
    }
    public static void main(String[] args) {
         //为事件分发线程预订一个工作：创建并显示本程序的 GUI
         javax.swing.SwingUtilities.invokeLater(new Runnable() {
             public void run() {
               UIManager.put("swing.boldMetal", Boolean.FALSE);
               new LabelDemo ().createAndShowGUI();
             }
         });
    }
}
```

编译并运行程序，结果如图 14.19 所示。

从图 14.19 中可以看出，所有的标签内容在垂直方向上都有默认的对齐方式：居中对齐。最上面的标签，包含有文本和图标，水平方向上居中显示。第二个标签只包含图标，对于只包含图标的标签，默认水平居中显示。最下面的标签只含有文本，默认靠左对齐。



图 14.19　标签 JLabel 组件的三种形式

14.3.2　进度条组件 JProgressBar

有时，在一个程序中运行的任务可能会持续一些时间才能完成。用户友好的程序应该对用户一些提示，以说明任务可能需要多长时间，当前已经完成了多少等。一种方式是使用动画图像来说明，另外一种方式是设置等待状态鼠标（使用 Cursor 类和组件的 setCursor() 方法）。例如，下面的代码设置当鼠标位于一个容器之上时显示等待状态鼠标。

```
Container.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR);
```

可以使用如图 14.20 所示的进度条来传达任务进度信息。

有时，对于处理时间比较长的任务，可能不能确定它具体要用多长时间才能完成，这时，可以将进度条置于“不确定模式”，能显示处理状态，但不能显示完成多少的信息。



处理不确定模式的进度条显示动画以说明工作正在进行。一旦进度条能显示足够的信息，就要将其切换为默认模式（确定模式）。不确定模式如图 14.21 所示。

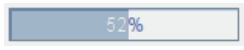


图 14.20　进度信息



14.21　不确定模式

Swing 提供了三个类来帮助在程序中使用进度条，分别为 JProgressBar、ProgressMonitor 和 ProgressMonitorInputStream。本书只介绍 JProgressBar 的使用。这三个类简单说明如下。

- JProgressBar：一个可视化的组件，用来图形化地显示总任务已经完成了多少。
- ProgressMonitor：不是一个可视化的组件。这个类的实例监视任务的完成情况并且如果必要的话弹出一个对话框。
- ProgressMonitorInputStream：获得一个输入流的监视器，通常用来监视读取文件的进度。

1．使用确定进度条

下面的代码片段显示了如何创建和设置进度条：

```
//声明成员变量
JProgressBar progressBar;
……
//构造器中
progressBar = new JProgressBar(0, task.getLengthOfTask());
progressBar.setValue(0);
progressBar.setStringPainted(true);
```

JProgressBar 的构造器有两个参数，分别代表进度条的最小值和最大值。还可以通过使用 setMinimum()和 setMaximum()来设置这些值。在这里分别为 0 和任务的长度。然而，进度条的最小值和最大值可以是任何值，甚至是负值。方法 setValue(0)将进度条的当前值设为 0。

调用方法 setStringPainted()能使进度条显示任务已经完成的百分比。默认情况下，进度条以百分比的格式显示其方法 getPercentComplete()方法返回的值，如 33%。也可以通过调用 setString()方法来替代默认的字符串。例如：

```
if(…)
    progressBar.setString("进度过半了！");
```

2．使用不确定进度条

通过调用 JProgressBar 的 setIndeterminate(true)方法，可以将进度条设为不确定进度条。如果将参数改为“false”，则将进度条设为确定进度条。

14.3.3　分隔线组件 JSeparator

JSeparator 类提供水平或垂直方向上的分隔线或空的区域，在菜单和工具栏中是最经常使用的。实际上，甚至在不知道有 JSeparator 类存在的情况下都可以使用分隔线，因为菜单和工具栏提供了很方便的方法来创建和添加分隔线。向菜单中添加菜单项和分隔线非常简单，如以下代码所示。

```
menu.add(menuItem1);
```

```
menu.add(menuItem2);
menu.add(menuItem3);
menu.addSeparator();
menu.add(rbMenuItem1);
menu.add(rbMenuItem2);
menu.addSeparator();
menu.add(cbMenuItem1);
menu.add(cbMenuItem2);
menu.addSeparator();
menu.add(submenu);
```

向工具栏中添加分隔线与此相似。可以在许多容器中直接使用 JSeparator 类来提供一条分隔线。JSeparator 类几乎没有什么 API，而且相当容易使用，只要记住以下原则：在大多数实现中，垂直分隔线的首选高度为 0，水平分隔线的首选宽度为 0。这意味着，只有在设置了分隔线的首选大小后，分隔线才会是可见的。

另外，还可以将分隔线置于布局管理器的控制之下（如 BorderLayout 或 BoxLayout），布局管理器会对其进行拉伸并填满其可用的显示区域。只有分隔线的高度和宽度都不为 0 时，才会绘制。创建并添加一个垂直分隔线的代码如下所示。

```
panel.add(new JSeparator(JSeparator.VERTICAL));
```

也可以先创建一个 JSeparator 对象，然后使用 void setOrientation(int)方法设置其方向。例如：

```
JSeparator sp = new JSeparator( );
sp.setOrientation(SwingConstants.HORIZONTAL);      //设置为水平分隔线
// sp.setOrientation(SwingConstants.VERTICAL);     //设置为垂直分隔线
```

在默认情况下，创建的分隔线是水平分隔线。

【例 14.12】使用分隔线的完整示例代码。

```
package chapter14;
import javax.swing.*;
import java.awt.GridLayout;
public class SeparatorDemo {
     JFrame frame;                                  //声明窗体对象
     JPanel contentPane;                            //声明面板对象
     JLabel label1,label2,label3;                   //声明三个标签对象
     JSeparator sp1,sp2;                            //声明两个分隔线对象
     public SeparatorDemo () {                      //不带参数的构造器
         frame = new JFrame("SeparatorDemo ");      //创建窗体对象
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         contentPane = new JPanel(new GridLayout(1,5));//1 行 5 列的面板
         frame.setContentPane(contentPane);         //设置内容面板
         label1 = new JLabel("标签 1");             //创建第一个标签对象
         label2 = new JLabel("标签 2");             //创建第二个标签对象
         label3 = new JLabel("标签 3");             //创建第三个标签对象
         sp1 = new JSeparator(SwingConstants.VERTICAL); //创建垂直的分隔线
         sp2 = new JSeparator(SwingConstants.VERTICAL);
     }
     // 创建 GUI 界面并显示
     public void createAndShowGUI() {
         //添加组件
```

```
            contentPane.add(label1);
            contentPane.add(sp1);
            contentPane.add(label2);
            contentPane.add(sp2);
            contentPane.add(label3);
            //显示窗口
            frame.pack();
            frame.setVisible(true);
        }
        public static void main(String[] args) {
            //为事件分发线程预订一个工作：创建并显示本程序的 GUI
            javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                  new SeparatorDemo ().createAndShowGUI();
                }
            });
        }
}
```



图 14.22　使用垂直分隔线

编译并运行程序，结果如图 14.22 所示。

在这个程序中，创建了两个垂直分隔线。在构造分隔线时，通过构造器的参数 SwingConstants.VERTICAL 指定分隔线为垂直的。如果不指明的话，默认情况下为水平分隔线。

14.3.4　工具提示组件 JToolTip

在 Swing 中，很容易就能为任何 JComponent 对象创建一个工具提示。使用 setToolTipText 方法为组件设置工具提示。例如，向三个按钮添加工具提示，只需要使用如下三行代码即可：

```
b1.setToolTipText("单击这个按钮使中间按钮不可用。");
b2.setToolTipText("单击这个按钮时无响应。");
b3.setToolTipText("单击这个按钮使中间按钮可用。");
```

当用户将鼠标悬停在任一个按钮上时，相应的提示就会出现，如图 14.23 所示。



图 14.23　显示工具提示

要在程序中关闭工具提示，使用 setToolTipText(null)方法。

14.4　小结

本章主要介绍 JComponent 类和 Swing 的基本组件。JComponent 类是所有 Swing 中非顶层组件的直接或间接超类，它定义了公共的 API，所以了解 JComponent 类有助于掌握其他 Swing 组件的使用。

在学习本章内容时，重点掌握 Swing 包中各种组件的使用。学习本章并不难，但是读者一定要将书中的示例程序亲手来编码、调试和运行，以掌握各种组件的使用。下一章向读者介绍 Swing 中的高级组件。

第 15 章　使用 Swing 高级组件

除了基本控件之外，Swing 还提供了很多高级组件，以帮助程序员创建更加复杂但更加有表现力的应用程序。这些高级组件包括可以交互式显示高度格式化信息的控件，如颜色选择器、表和树等，也包括非顶层的容器组件和具有特殊用途的窗口组件。



15.1　交互式显示高度格式化信息控件

交互式显示高度格式化信息控件是这样一些组件，它们可以被用来显示高度格式化的信息，这些信息也可以被用户修改。Swing 中这样的组件有颜色选择器 JColorChooser、文件选择器 JFileChooser、文本编辑组件 JEditorPane 和 JTextPane、文本区组件 JTextArea、表组件 JTable 和树组件 JTree。

15.1.1　颜色选择器 JColorChooser

JColorChooser 类代表一个颜色选择器（或叫调色板），用户可以从中选择颜色。可以将颜色选择器放在程序 GUI 中的任何地方。使用 JColorChooser 的 API 还能很容易地弹出包含颜色选择器的对话框（模态或非模态）。创建一个颜色选择器可以使用下面的代码：

```
JColorChooser colorChooser = new JColorChooser( );
```

在使用不带参数的构造器创建一个 JColorChooser 对象时，选择器默认显示（或选中）的颜色是 Color.white。



这里的 Color 的全限定名是 java.awt.Color，Color 类用于封装默认 sRGB 颜色空间中的颜色。sRGB 是一个被推荐使用的标准 RGB 颜色空间。在 Color 类中定义了很多静态常量，用来代表颜色。例如，Color.yellow 代表黄色，Color.green 代表绿色，Color.red 代表红色。更多的信息请参阅 Java API 文档。

也可以创建一个初始化为指定颜色的 JColorChooser 对象，如以下代码所示。

```
JColorChooser colorChooser = new JColorChooser( Color.green);
```

这里指明创建的颜色选择器初始选定颜色为绿色。要获取颜色选取器的当前颜色值，



使用其 getColor()方法，返回一个 Color 对象。例如：

```
Color selectedColor = colorChooser.getColor( );//获得在颜色选择器中选择的颜色对象
JLabel label = new JLabel("颜色");                    //创建标签对象
label.setForeground(selectedColor);//将标签的字体颜色设为从标签选择器中选择的颜色
```

还可以使用 showDialog 方法在对话框中显示颜色选择器。showDialog 方法的定义如下所示。

```
static Color showDialog(Component component, String title, Color initialColor)
```

其中第一个参数是对话框组件的父组件，第二个参数是对话框的标题，第三个参数是初始选择的颜色。该方法的返回值是一个代表所选择颜色的 Color 对象。显示的颜色选择器对话框如图 15.1 所示。

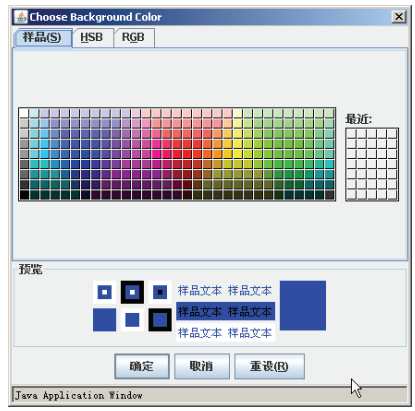


图 15.1　颜色选择对话框

在三种情况下，颜色对话框会消失：用户选择了一个颜色并单击“确定”按钮，用户单击“取消”按钮撤销操作，或者用户使用窗体关闭按钮关闭对话框。如果用户选择一个颜色，showDialog()方法返回新的颜色。如果用户取消操作或关闭窗口，该方法返回 null。

由 showDialog()方法创建的对话框是模态对话框。如果想要一个非模态对话框，可以使用 JColorChooser 类的 createDialog()方法。默认情况下，颜色选择器带有一个预览面板。要移除预览面板，使用 setPreviewPanel()方法，代码如下所示。

```
colorChooser.setPreviewPanel(new JPanel());
```

因为一个空白的 JPanel 对象没有大小和默认的视图，所以将颜色选择器的预览面板设为空白的面板实际上是将其移除了。如果想重新将预览面板设为默认的，使用 null 作为 setPreviewPanel()的参数，如下所示。

```
colorChooser.setPreviewPanel(null);
```

15.1.2　文件选择器 JFileChooser

文件选择器提供了一个用于文件系统导航的 GUI。在文件选择器中，用户既可以选择一个文件或目录，也可以输入一个文件或目录的名称。使用 JFileChooser 的 API 来显示一个包含文件选择器的模态对话框。也可以通过添加一个 JFileChooser 的实例到一个容器当中实现一个文件选择器。

JFileChooser API 提供打开文件对话框和保存文件对话框。这两个对话框看上去几乎一样，只是对话框窗口的标题和按钮上的文本不同。如图 15.2 和图 15.3 所示。

创建一个标准的打开文件对话框只需要两行代码，如下所示。

```
//创建一个文件选择器
final JFileChooser fc = new JFileChooser( );
……
//调用 JFileChooser 的 API，打开文件对话框
int returnVal = fc.showOpenDialog(Component);
```

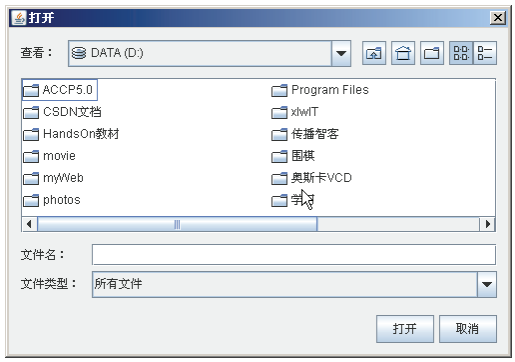


图 15.2　打开文件对话框

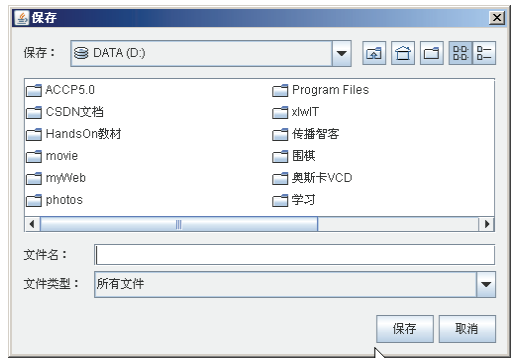


图 15.3　保存文件对话框

传递给 showOpenDialog()方法的参数代表对话框的父组件。父组件影响文件打开对话框的位置，也影响对话框所依赖的窗体。如果父组件位于窗体内，那么该对话框依赖于窗体。当窗体被最小化时对话框消失，当窗体最大化时，对话框重新出现。

默认情况下，文件选择器显示的是主目录。可以使用 JFileChooser 的其他构造器来指定文件选择器的初始目录，或者使用 setCurrentDirectory()方法指定目录。

showOpenDialog()和 showSaveDialog()方法返回一个整数，用来说明用户是否选择了一个文件。如果返回值是 APPROVE_OPTION，说明用户选择了一个文件；如果是其他值，则没有选择文件。经常使用下面这种形式的代码进行判断：

```
final JFileChooser fc = new JFileChooser( );    //创建文件选择器对象
……
int returnVal = fc.showOpenDialog(Component);  //显示文件选择器
if(returnVal == JFileChooser.APPROVE_OPTION){
    File file = fc.getSelecteFile( );           //获得在文件选择器选择的文件
    ……     //这里是真正打开文件的代码
}
```

要获得所选择的文件（或目录，如果设置文件选择器只允许选择目录），调用文件选择器的 getSelectedFile 方法，该方法返回一个 File 文件对象的实例。

一个 JFileChooser 对象只代表选择文件的 GUI 界面，它并不能真正打开或保存文件。真正打开或保存文件还需要另外编写代码，使用基本的文件 I/O 流来完成。调用 showSaveDialog()方法可以出现一个保存文件对话框，如以下代码所示。

```
int returnVal = fc.showSaveDialog(Component);
```

使用同一个 JFileChooser 实例来显示打开和保存文件对话框，程序还可以获得如下的好处：

- 文件选择器会记忆两次使用之间的当前目录，所以打开和保存时会自动地共享相同的当前目录。
- 只需要自定义一个文件选择器，那么自定义部分会同时应用于打开和保存对话框。

默认的文件选择模式是 FILES_ONLY。通过改变文件选择器的选择模式，使得文件选择器只能显示和选择文件目录。例如，下面的代码设置文件选择器 fc 只能选择目录，而不能选择文件：

```
fc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
```



如果想文件和目录都可以选择，参数使用 FILEA_AND_DIRECTORIES。例如：

```
fc.setFileSelectionMode(JFileChooser. FILEA_AND_DIRECTORIES);
```

如果要自定义对话框，如自定义文件对话框的标题和按钮上显示的文字，使用如下的代码。

```
//创建一个文件选择器
final JFileChooser fc = new JFileChooser( );
……
//调用 JFileChooser 的 API，打开文件对话框
int returnVal = fc.showOpenDialog(Component, String);
```

其中 showOpenDialog 方法的第二个参数是一个字符串，它会显示在文件对话框的标题和按钮上。

15.1.3　文本编辑组件 JEditorPane 和 JTextPane

Swing 中有两个类支持样式文本：JEditorPane（编辑器窗格）及其子类 JTextPane（文本窗格）。JEditorPane 类是 Swing 样式文本组件的基础，并提供一种机制为自定义文本格式提供支持。

javax.swing.JEditorPane 是 javax.swing.text.JTextComponent 类的子类，它是一个可以编辑各种内容的文本组件。创建一个编辑器窗格，使用如下的代码：

```
JEditorPane editorPane = new JEditorPane( );
```

这时会创建一个空白的 JEditorPane 对象，默认情况下是可编辑的。如果只需要显示信息，不需要编辑，使用方法 setEditable(false)将其设为不可编辑的。

```
editorPane.setEditable(false);
```

要获得用户在编辑器窗格中输入的文字，使用如下方法：

```
String input = editor.getText();
```

在实际使用时，通常会把一个 JEditorPane 对象添加到一个滚动面板中，这样当编辑的内容超出可编辑的区域时，可以滚动查看。例如：

```
JScrollPane editorScrollPane = new JScrollPane(editorPane);
```

JTextPane 是 JEditorPane 的子类，它可以将组件和图像嵌入到文本中。JEditorPane 默认情况下可以读、写和编辑纯文本、HTML 文本和 RTF 文本。而 JTextPane 也继承了这种能力，但是加入了一定的限制。创建和初始化一个 JTextPane 对象实例，使用如下的代码：

```
JTextPane textPane = new JTextPane( );
```

【例 15.1】使用了 JEditorPane 和 JTextPane 对象的示例程序。

```
package chapter15;
import javax.swing. JEditorPane;
import javax.swing. JTextPane;
import javax.swing. JScrollPane;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.Dimension;
public class TextSamplerDemo {
```

```
        JEditorPane editorPane;     //声明一个 JEditorPane 类型变量
        JTextPane textPane;         //声明一个 JTextPane 类型变量
        JScrollPane scrollPane1,scrollPane2;   //声明两个 JScrollPane 类型变量
        JFrame frame;               //声明一个 JFrame 类型变量
        JPanel contentPane;         //声明一个容纳文本组件的面板，并指定其为内容面板
        JLabel label1,label2;
        public TextSamplerDemo () {             //不带参数的构造器
           //创建并设置窗口.
           frame = new JFrame("TextSamplerDemo");
           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
           contentPane = new JPanel();
           frame.setContentPane(contentPane);
           editorPane = new JEditorPane( );              //创建 JEditorPane 对象
           textPane = new JTextPane( );                  //创建 JTextPane 对象
           scrollPane1 = new JScrollPane(editorPane);  //创建滚动面板
           scrollPane1.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_
SCROLLBAR_ALWAYS);
           scrollPane1.setPreferredSize(new Dimension(250, 145));
                                                   //设置滚动面板的首选大小
           scrollPane1.setMinimumSize(new Dimension(10, 10));
           scrollPane2 = new JScrollPane(textPane);//创建包含文本区的滚动面板

scrollPane2.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAY
S);
           scrollPane2.setPreferredSize(new Dimension(250, 145));//设置首选大小
           scrollPane2.setMinimumSize(new Dimension(10, 10));
           label1 = new JLabel("JEditorPane:");          //创建标签对象
           label2 = new JLabel("JTextPane:");
        }
        //创建 GUI 界面并显示
        private void createAndShowGUI() {
           //将两个文本组件添加到窗体的内容面板中
           contentPane.add(label1);
           contentPane.add(scrollPane1);
           contentPane.add(label2);
           contentPane.add(scrollPane2);
           //显示窗口
           frame.pack();
           frame.setVisible(true);
        }
        public static void main(String[] args) {
           //为事件分发线程预订一个工作：创建并显示本程序的 GUI
           javax.swing.SwingUtilities.invokeLater(new Runnable() {
              public void run() {
                 new TextSamplerDemo().createAndShowGUI();
              }
           });
        }
    }
```

编译并运行此程序，运行结果如图 15.4 所示。

在上面的这个程序中，创建了一个 JEditorPane 对象和一个 JTextPane 对象，一般情况下，它们都要位于滚动面板中，当内容超出显示区域时，会自动出现滚动条。默认情况下，这两个组件都是可编辑的。



图 15.4　JEditorPane 和 JTextPane

15.1.4　文本区组件 JTextArea

JTextArea 类可以显示多行文本，并且允许用户编辑文本。例如，在本书的很多示例程序中，使用 JTextArea 来显示程序的输出。创建并初始化 JTextArea 的代码如下所示。

```
JTextArea textArea = new JTextArea(5,20);
JScrollPane scrollPane = new JScrollPane(textArea);
textArea.setEditable(false);
```

在上面的代码片段中，第一行声明并创建一个 JTextArea 对象，构造器中的两个整型参数分别代表文本区的行数和列数（即文本区显示的大小）。第二行，创建一个滚动面板，将文本区对象作为 JScrollPane 构造器的参数，可以将文本区放入滚动面板。否则，文本区不会自动滚动。第三行，调用 JTextArea 对象的 setEditable 方法，通过参数 false 将其设为不可编辑的（即只能显示、选择和复制信息，但不能直接修改或输入信息）。默认情况下，文本区是可编辑的。

下面的代码用来向文本区中添加文本：

```
private final static String newline = "\n";//定义常量字符串 newline，它代表换行符
...
textArea.append(text + newline);    //将 text 所代表的文本和换行符添加到文本区中
```

插入文本后，文本区会自动滚动使新添加的文本可见。也可以在调用 append()方法后，通过移动插入点到文本区的结尾处来强制文本区滚动到文本区的底部。例如：

```
textArea.setCaretPosition(textArea.getDocument().getLength());
```

一个文本区只能显示一种字体和一种颜色，但是可以设置使用哪种字体和颜色。另外，在默认情况下，文本区中的文本是不自动换行的，所有文本都在一行内，如果有滚动窗格的话，会出现水平滚动条。可以通过 setLineWrap()方法指定文本区中的文本可以自动换行，并通过 setWrapStyleWord()方法指定在单词中间换行而不是在字符中间换行。

【例 15.2】创建一个完整的示例程序，在程序中初始化一个可编辑的文本区，并设置文本区使用指定的斜体字，并且内容在单词中间换行。

```
package chapter15;
import javax.swing. JTextArea;
import javax.swing. JScrollPane;
import javax.swing.JPanel;
import javax.swing.JFrame;
import java.awt.Dimension;
public class TextAreaDemo{
    JTextArea textArea;                    //声明一个 JTextArea 类型变量
    JScrollPane scrollPane;                //声明一个 JScrollPane 类型变量
    JFrame frame;                          //声明一个 JFrame 类型变量
```

```
    JPanel contentPane;          //声明一个容纳文本组件的面板，并指定其为内容面板
    public TextAreaDemo () {
       //创建并设置窗口
       frame = new JFrame("TextAreaDemo ");
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       contentPane = new JPanel();
       frame.setContentPane(contentPane);              //指定内容面板
       String content = "文本区是一个\"纯文本\"组件，" +
              "这意味着虽然它能以任何字体显示文本，" +
              "但所有的文本都以同样的字体显示。";
       textArea = new JTextArea(5,20,content);      //创建 JTextPane 对象
       textArea.append(content);              //将字符串 content 添加到文本区中
       textArea.setFont(new Font("宋体",Font.ITALIC,16));
                                              //设置字体：宋体，斜体，16 号字
       textArea.setLineWrap(true);            //设置文本区中的文本可以自动换行
       textArea.setWrapStyleWord(true);       //设置文本区中文本可以在单词中间换行
       scrollPane = new JScrollPane(textArea);      //创建滚动面板
    }
    //创建 GUI 界面并显示
    private void createAndShowGUI() {
       contentPane.add(scrollPane);           //将文本区组件添加到窗体的内容面板中
       //显示窗口
       frame.pack();
       frame.setVisible(true);
    }
    public static void main(String[] args) {
       //为事件分发线程预订一个工作：创建并显示本程序的 GUI
       javax.swing.SwingUtilities.invokeLater(new Runnable() {
          public void run() {
             new TextAreaDemo().createAndShowGUI();
          }
       });
    }
}
```

编译并运行此程序，运行结果如图 15.5 和图 15.6 所示。

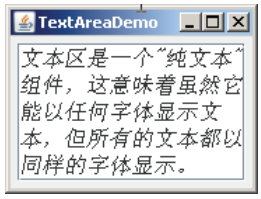


图 15.5　显示中文的文本区

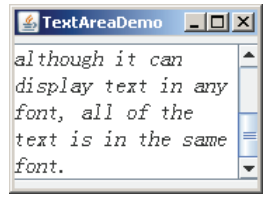


图 15.6　显示英文的文本区

在上面的程序中，通过调用 JTextArea 对象的 setFont()方法设置字体，调用 JTextArea 对象的 setLineWrap(true)方法设置其文本可以自动换行，调用 JTextArea 对象的 setWrapStyleWord(true)方法设置在单词中间换行。从图 15.6 中可以看出，英文文本在单词中间换行。

15.1.5　表组件 JTable

使用 JTable 类，可以以表格的形式显示数据，也允许用户编辑数据。JTabel 本身并不包含或缓存数据，它只是简单地作为数据的显示视图。图 15.7 为一个 JTable 对象示例。



图 15.7　JTable 对象

【例 15.3】创建如图 15.7 所示的表格的示例程序如下，在程序中创建并初始化一个简单的表格。

```
package chapter15;
import javax.swing. JTable;
import javax.swing. JScrollPane;
import javax.swing.JPanel;
import javax.swing.JFrame;
public class TableDemo{
    JTable table;                                   //声明一个 JTable 类型变量
    JScrollPane scrollPane;                         //声明一个 JScrollPane 类型变量
    JFrame frame;                                   //声明一个 JFrame 类型变量
    JPanel contentPane;          //声明一个容纳文本组件的面板，并指定其为内容面板
    public TableDemo() {
       //创建并设置窗口
       frame = new JFrame("TableDemo");
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       contentPane = new JPanel();
       frame.setContentPane(contentPane);   //指定内容面板
       String[ ] columnNames = "姓名","性别","年龄","家庭住址","婚否";
                                                    //声明表的列名
       Object[ ][ ] data = {                        //声明表中的数据，存放在二维数组中
            {" 张 小 明 "," 男 ",new  Integer(18)," 河 南 省 郑 州 市 ",new
Boolean(false)},
            {"李浩","男",new Integer(28),"北京市",new Boolean(true)},
            {" 程 小 娜 "," 女 ",new  Integer(21)," 河 北 省 石 家 庄 市 ",new
Boolean(false)},
            {"王金柱","男",new Integer(38), "广东省广州市",new Boolean(true)}
       };
       table = new JTable(date,columnName); //使用列名和数据初始化一个表格对象
       scrollPane = new JScrollPane(table); //创建滚动面板
       table.setFillsViewportHeight(true);  //设置表格的高度填满滚动窗格的高度
    }
    //创建 GUI 界面并显示
    private void createAndShowGUI() {
       contentPane.add(scrollPane);         //将文本区组件添加到窗体的内容面板中
       //显示窗口
       frame.pack();
       frame.setVisible(true);
    }
    public static void main(String[] args) {
       //为事件分发线程预订一个工作：创建并显示本程序的 GUI
       javax.swing.SwingUtilities.invokeLater(new Runnable() {
          public void run() {
```

```
                new TableDemo().createAndShowGUI();
            }
        });
    }
}
```

在上面的代码中，在一个字符串数组中声明表的列名，如下所示。

```
String[ ] columnNames = "姓名","性别","年龄","家庭住址","婚否";
```

表中的数据在一个二维 Object 数组中被初始化并存储的，如以下代码所示。

```
Object[ ][ ] data = {
            {"张小明","男",new Integer(18),"河南省郑州市",new Boolean
(false)},
            {"李浩","男",new Integer(28),"北京市",new Boolean(true)},
            {"程小娜","女",new Integer(21),"河北省石家庄市",new Boolean
(false)},
            {"王金柱","男",new Integer(38), "广东省广州市",new Boolean(true)}
        };
```

然后使用这些数据和表的列名构造一个 JTable 实例，代码如下。

```
table = new JTable(date,columnName);
```

JTable 的构造器中有两种可以直接接收数据：

```
JTable(Object[ ][ ] rowData, Object[] columnNames)
JTable(Vector rowData, Vector columnNames)
```

这两个构造器简单而且很容易使用。但是，它们也有缺点：

- 这样构造的表格，每一个单元格都是可编辑的。
- 这样构造的表格，将所有的数据类型都当做一样的（String 类型）。
- 这样构造的表格，需要将所有的数据都先存储在数组或向量中，如果要以表格显示从数据库中查询的数据，还需要先将查询到的数据复制到数组或向量中，会降低程序的效率。

构造好 JTable 对象以后，需要创建一个滚动窗格，作为 JTable 对象的容器：

```
scrollPane = new JScrollPane(table);
table.setFillsViewportHeight(true);
```

将 JTable 对象作为 JScrollPane 构造器的参数，将创建一个滚动窗格作为 JTable 对象的容器，JTable 对象会自动地被添加到容器中。JTable.setFillsViewportHeight()用来设置 fillsViewportHeight 属性。当此属性为 true 时，JTable 对象会使用容器的整个高度，即使 JTable 对象没有足够多的行来填充整个垂直空间。这很容易将 JTable 对象作为拖放对象使用。

默认情况下，表中的所有列都是等宽的，并且所有的列自动填充表格的整个宽度。当调整表格的宽度时，所有的列都同步地改变宽窄。

要想自定义初始的列宽，可以调用 TableColumn 对象（代表每一列）的 setPreferredWidth()方法。例如在下面的代码中，设置第三列的宽度比其他列的宽度宽。

```
TableColumn column = null;
for(int i = 0 ; i<5; i++){
    column = table.getColumnModel().getColumn(i);
                                        //获取代表每一列的 TableColumn 对象
    if(i = = 2){
```



```
            column.setPreferredWidth(100);          //第三列更宽
        }else{
            column.setPreferredWidth(50);           //其他各列都比第三列要窄
        }
    }
```

如代码中所示，表中的每一列都由一个 TableColumn 对象代表。TableColumn 有相应的 get 和 set 方法用于获取或设置某一列的最小、最大和首选宽度，还提供获取当前宽度的方法。

15.1.6　树组件 JTree

使用 JTree 类，可以显示分层级的数据。一个 JTree 对象并不真正包含数据；它简单地提供数据的一个视图。与任何高级 Swing 组件一样，树也是通过查询其数据模型来获得数据的。图 15.8 显示了一个树的结构：

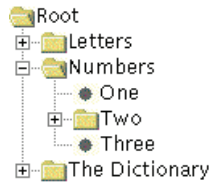


图 15.8　树状目录

正如图 15.8 所示，JTree 垂直显示数据。每一行包含一个数据项，称为“节点（node）”。每一个树有一个“根节点（root node）”，所有另外的节点都继承自根节点。一个节点既可以有子节点，也可以没有子节点。可以有子节点的节点称为枝节点（不论当前有没有子节点）。不能有子节点的节点称为叶子节点。

枝节点可以有任意数量的子节点。典型地，通过单击，用户可以展开和收缩枝节点，从而使其子节点可见或隐藏。默认情况下，除了根节点，所有的节点刚开始都是隐藏的。

树中一个指定的节点既可以通过一个 TreePath（一个封装了节点及其所有父节点的对象）标识，也可以通过显示的行（在显示区每一行显示一个节点）标识。创建一个节点对象，使用下面的代码。

```
DefaultMutableTreeNode top = new DefaultMutableTreeNode("同学录");
```

在上面的代码中，创建了一个节点 top。DefaultMutableTreeNode 类位于 javax.swing.tree 包中，它的一个实例对象，就代表一个节点。DefaultMutableTreeNode 类的构造器所带的参数字符串代表节点的名称。使用 DefaultMutableTreeNode 创建另外三个子节点 second1、second2 和 second3，使用如下的代码：

```
DefaultMutableTreeNode second1 = new DefaultMutableTreeNode("小学同学");
DefaultMutableTreeNode second2 = new DefaultMutableTreeNode("中学同学");
DefaultMutableTreeNode second3 = new DefaultMutableTreeNode("大学同学");
```

将 top 节点作为根节点，将 second1、second2 和 second3 作为 top 的子节点，使用如下的代码：

```
top.add(second1);       //将节点 second1 添加到 top 节点
top.add(second2);       //将节点 second2 添加到 top 节点
top.add(second3);       //将节点 second3 添加到 top 节点
```

每一个 DefaultMutableTreeNode 类的实例对象，就代表一个节点。一棵树中所有的节点都是一个个的 DefaultMutableTreeNode 的实例对象。使用 DefaultMutableTreeNode 类的 add() 方法，将一个子节点添加到另一个子节点上，从而两个节点形成父子关系。依次创建和添加节点，就形成了节点层级。要创建一个包含上面创建的节点层级的树，使用下面的代码：

```
JTree tree = new JTree(top);                    //创建含有 top 作为根节点的树
JScrollPane treeView = new JScrollPane(tree);//将创建的 JTree 对象放入滚动窗格中
```

指定根节点作为 JTree 构造器的参数，从而创建一个包含所有节点的树。最后，将所创建的树放入一个滚动窗格，这样当展开树的节点时，就可以通过滚动查看树中的节点。

【例 15.4】使用 JTree 对象的示例程序。

```
package chapter15;
import javax.swing. JTree;
import javax.swing.tree. DefaultMutableTreeNode;
import javax.swing. JScrollPane;
import javax.swing.JPanel;
import javax.swing.JFrame;
public class TreeDemo{
    JTree tree;                  //声明一个 JTable 类型变量
    JScrollPane scrollPane;      //声明一个 JScrollPane 类型变量
    JFrame frame;                //声明一个 JFrame 类型变量
    JPanel contentPane;          //声明一个容纳文本组件的面板，并指定其为内容面板
    public TreeDemo () {
       //创建并设置窗口
       frame = new JFrame("TreeDemo ");
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       contentPane = new JPanel();
       frame.setContentPane(contentPane);  //指定内容面板
       //以下代码创建树中的各层级节点
       DefaultMutableTreeNode top = new DefaultMutableTreeNode("同学录");
                                           //创建根节点
       DefaultMutableTreeNode second1 = new DefaultMutableTreeNode("小学
同学");                                    //创建第一级子节点
       DefaultMutableTreeNode second2 = new DefaultMutableTreeNode("中学
同学");                                    //创建第一级子节点
       DefaultMutableTreeNode second3 = new DefaultMutableTreeNode("大学
同学");                                    //创建第一级子节点
       top.add(second1);                   //将节点 second1 添加到 top 节点
       top.add(second2);                   //将节点 second2 添加到 top 节点
       top.add(second3);                   //将节点 second3 添加到 top 节点
       DefaultMutableTreeNode name1 = new DefaultMutableTreeNode("张小明");
                                           //创建第二级子节点
       second1.add(name1);                 //将张小明添加到小学同学节点下
       DefaultMutableTreeNode name2 = new DefaultMutableTreeNode("刘芳");
                                           //创建第二级子节点
       second2.add(name2);                 //将刘芳添加到中学同学节点下
       DefaultMutableTreeNode name3 = new DefaultMutableTreeNode("老赵");
                                           //创建第二级子节点
       second3.add(name3);                 //将老赵添加到大学同学节点下
       DefaultMutableTreeNode phone1 = new DefaultMutableTreeNode ("63122166");
                                           //创建第三级子节点
       name1.add(phone1);                  //将电话添加到张小明节点下
       DefaultMutableTreeNode phone2 = new DefaultMutableTreeNode ("63526666");
                                           //创建第三级子节点
       name2.add(phone2);                  //将电话添加到刘芳节点下
       DefaultMutableTreeNode phone3 = new DefaultMutableTreeNode("67834353");
                                           //创建第三级子节点
       name3.add(phone3);                  //将电话添加到张小明节点下
```



```
        //使用一个包含上面所创建节点的树
        tree = new JTree(top);                                    //创建树对象
        scrollPane = new JScrollPane(tree);                       //创建滚动面板
    }
    //创建 GUI 界面并显示
    private void createAndShowGUI() {
        contentPane.add(scrollPane);      //将文本区组件添加到窗体的内容面板中
        //显示窗口
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作：创建并显示本程序的 GUI
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new TreeDemo().createAndShowGUI();
            }
        });
    }
}
```

编译并运行上述程序，运行结果如图 15.9 所示。



图 15.9　使用 JTree 对象

在上面的程序中，创建了三级的树结构层次，以默认的外观显示。另外，也可以自定义树的显示。例如，可以不显示根节点，使用如下代码。

```
tree.setRootVisible(false);
```

要显示节点，将此方法的参数改为 true。在默认情况下，树中的顶层节点（如果根节点不可见则为子节点，否则为根节点）不显示操作句柄。可以使用下列代码显示其操作句柄：

```
tree.setShowsRootHandles(true);
```

默认情况下，各节点之间有角线连接。可以使用方法 putClientProperty()来设置 JTree.lineStyle 客户端属性，来设置是否显示角线或怎样显示。

```
tree.putClientProperty("JTree.lineStyle", "Horizontal");
                                                    //设置只使用水平线分组节点
tree.putClientProperty("JTree.lineStyle", "None");      //设置不使用角线
```

图 15.10 和图 15.11 分别为设置不同的 JTree.lintStyle 属性后的显示。



图 15.10　使用水平线



图 15.11　不使用角线

将图 15.9、图 15.10 和图 15.11 相对比，可以很明显地看到显示的不同。默认情况下，树中的节点是不可编辑的。可以通过方法 setEditable(true)，将树中的节点设置为可编辑的。

```
    tree.setEditable(true);
```

这时再编译并运行程序，双击其中任一节点，就会发现光标可以插入节点文本中。这时修改文本，并按回车键确定，会发现节点文本内容已经被改变了。

使用上述方法创建的是一棵静态树，不能动态改变节点。要想创建一个可以动态添加和移除树节点或编辑节点上的文本，需要使用 DefaultTreeModel 对象来创建一棵可以动态改变的树。JTree 类也是一个 MVC 结构的组件，而 DefaultTreeModel 就代表其中模型的一个实现。下面的代码创建并初始化一棵动态树。

```
    DefaultMutableTreeNode rootNode = new DefaultMutableTreeNode("同学录");
    DefaultTreeModel treeModel = new DefaultTreeModel(rootNode);
    JTree tree = new JTree(treeModel);
    tree.setEditable(true);
    tree.getSelectionModel().setSelectionMode(TreeSelectionModel.SINGLE_TRE
E_SELECTION);
    tree.setShowsRootHandles(true);
```

通过创建树的模型——DefaultTreeModel 类的实例，可以调用该模型所支持的所有方法，例如 insertNodeInto 方法可以插入一个节点。DefaultTreeModel 是接口 TreeModel 的一个实现类。使用带有 TreeModel 参数的构造器创建 JTree 对象。

在上面的代码中，通过调用 setEditable(true)方法，可以编辑树中的节点。当用户结束编辑节点文本时，模型会保证模型事件通知到所有的监听器，做进一步的响应处理。虽然 DefaultMutableTreeNode 类也有改变节点内容的方法，但是这种改变不会通知到监听器。所以应尽可能地使用 DefaultTreeModel。

15.2 非顶层容器控件

Swing 的非顶层容器控件是多用途容器，被用于大多数的 Swing 应用程序。它们可以容纳其他组件，所以称为容器控件。非顶层容器控件包括面板组件 JPanel、滚动面板（滚动窗格）JscrollPane、拆分面板（窗格）JsplitPane、选项卡面板 JTabbedPane 和工具栏 JToolBar。

15.2.1 面板组件 JPanel

JPanel 类为轻量级的组件提供通用目的的容器。默认情况下，JPanel 对象除了自己的背景色之外，不给任何组件添加颜色。不过可以自定义面板的边框及自定义面板。

默认情况下，面板是不透明的。不透明的面板可以作为内容面板，并能帮助有效地进行绘图。可以调用 setOpaque()方法改变一个面板的透明度。一个透明的面板不绘制背景，因此任何在它下面的组件都会透视出来。使用面板可以通过将边框围绕着一组组件，从而将相关的组件组织在一起。

和其他容器一样，面板使用布局管理器来设置组件的位置和大小。默认地，面板的布局管理器是流动布局，即 FlowLayout 的一个实例，这种布局将组件按顺序在一行上排列，如果一行放满的话，会自动地换行。

通过 setLayout()方法可以很容易地设置任何其他的布局管理器，或者创建面板时指定一个布局管理器。从性能上说，在创建面板对象时就指定布局管理器更佳，因为避免了不必要的创建 FlowLayout 对象。下面的代码演示了当创建面板时，怎样设置布局管理器：



```
JPanel p = new JPanel(new BorderLayout());      //这种方法首选
```

使用这种方式设置面板的布局管理器时，不能使用 BoxLayout，因为 BoxLayout 的构造器要求一个事先存在的容器。例如：

```
JPanel p = new JPanel();
p.setLayout(new BoxLayout(p,BoxLayout.PAGE_AXIS));
```

使用 add()方法向面板中添加组件。当布局管理器是 FlowLayout、BoxLayout、GridLayout 或 SpringLayout 时，通常使用带有一个参数的 add()方法。例如：

```
aFlowPanel.add(aComponent);
aFlowPanel.add(anotherComponent);
```

当布局管理器是 BorderLayout 时，需要在参数中指定被添加的组件在面板中的位置。例如：

```
aBorderPanel.add(aComponent,BorderLayout.CENTER);
aBorderPanel.add(anotherComponent,BorderLayout.PAGE_END);
```

【例 15.5】创建并设置面板的一个完整示例程序。

```
package chapter15;
import javax.swing.JPanel;
import javax.swing.JFrame;
public class PanelDemo{
    JFrame frame;                  //声明一个 JFrame 类型变量
    JPanel mainPane;               //声明一个容纳文本组件的面板，并指定其为内容面板
    public PanelDemo () {
       //创建并设置窗口
       frame = new JFrame("PanelDemo ");
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       //创建面板
       mainPane = new JPanel();
       mainPane.setLayout(new BoxLayout(mainPane, BoxLayout.PAGE_AXIS));
                                                          //设置面板的布局方式
       mainPane.setOpaque(true);                          //设置面板为不透明
       mainPane.setBackground(new Color(255, 0, 0));      //设置面板的背景色
       mainPane.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
                                                          //设置面板的边框
       mainPane.add(Box.createRigidArea(new Dimension(0, 5)));
       frame.setContentPane(mainPane);                    //指定内容面板
    }
    //创建 GUI 界面并显示
    private void createAndShowGUI() {
       //显示窗口
       frame.pack();
       frame.setVisible(true);
    }
    public static void main(String[] args) {
       //为事件分发线程预订一个工作：创建并显示本程序的 GUI
       javax.swing.SwingUtilities.invokeLater(new Runnable() {
          public void run() {
             new PanelDemo ().createAndShowGUI();
          }
       });
    }
}
```

编译并运行上述程序，运行结果如图 15.12 所示。

在上面的程序中，创建了一个简单的窗体，窗体中包含一个 JPanel 组件。JPanel 组件在 Java 图形界面程序设计中是最经常使用到的容器组件之一。通常用来实现复杂的中间布局。

15.2.2 滚动面板 JScrollPane

JScrollPane 类为组件提供一个可滚动的视图。创建一个滚动面板的代码量是非常小的。图 15.13 显示了一个滚动面板中的文本区。因为随着文本的增加，文本区的大小也相应地会增长，所以要使用滚动视图来查看。滚动窗格会根据需要自动地创建滚动条、重绘客户区等。

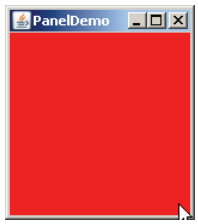


图 15.12 使用 JPanel 对象

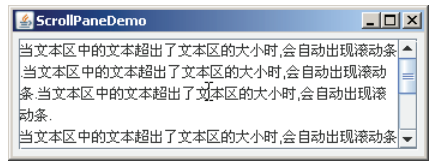


图 15.13 CheckBoxDemo.java 中的复选框

【例 15.6】创建使用滚动面板的示例程序，程序运行如图 15.13 所示。

```
package chapter15;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import java.awt.Dimension;
public class ScrollPaneDemo{
    JFrame frame;                                    //声明一个 JFrame 类型变量
    JPanel contentPane;                              //声明一个内容面板
    JTextArea textArea;                              //声明一个文本区
    JScrollPane scrollPane;                          //声明一个滚动窗格
    public ScrollPaneDemo () {
       //创建并设置窗口
       frame = new JFrame("ScrollPaneDemo ");
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       textArea = new JTextArea(5,30);               //创建文本区组件
       textArea.setLineWrap(true);
       scrollPane = new JScrollPane(textArea); //创建滚动窗格，包含文本区
       contentPane = new JPanel();
       contentPane.add(scrollPane);
       frame.setContentPane(contentPane);            //指定内容面板
    }
    //创建 GUI 界面并显示
    private void createAndShowGUI() {
       //显示窗口
       frame.pack();
       frame.setVisible(true);
    }
    public static void main(String[] args) {
       //为事件分发线程预订一个工作：创建并显示本程序的 GUI
       javax.swing.SwingUtilities.invokeLater(new Runnable() {
```



```
            public void run() {
                new ScrollPaneDemo().createAndShowGUI();
            }
        });
    }
}
```

使用下面的命令编译和运行此程序：

```
javac -d . ScrollPaneDemo.java
java chapter15. ScrollPaneDemo
```

在大多数情况下，对于要显示文本的组件，都需要封装在 JScrollPane 组件之中，为这些组件提供超出显示区域进行内容浏览的功能。

15.2.3　拆分面板 JSplitPane

JSplitPane 又称为拆分窗格，它显示两个组件，水平排列或上下排列。通过拖动窗格之间的拆分线，可以调节两个窗格的大小。还可以通过在拆分窗格中嵌套拆分窗格的方式，将屏幕空间分为三个或更多的组件。

不能将组件直接添加到一个拆分窗格中，而要先将每一个组件添加到一个滚动窗格中，然后将滚动窗格添加到拆分窗格中，如图 15.14 所示。



图 15.14　拆分窗格

创建拆分窗格，使用如下的代码。

```
    //创建带有两个滚动窗格的拆分窗格
    JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,scrollPane1,
scrollPane2);
    splitPane.setOneTouchExpandable(true);
    splitPane.setDividerLocation(150);
    //为拆分窗格中的两个滚动窗格提供最小的大小限制
    Dimension minimumSize = new Dimension(100,50) ;
    scrollPane1.setMinimumSize(minimumSize) ;
    scrollPane2.setMinimumSize(minimumSize) ;
```

在 JSplitPane 的构造器中，需要三个参数。第一个参数代表拆分的方向：水平还是垂直拆分。另外两个参数是放入拆分窗格的滚动窗格。在上面的代码中，指定拆分窗格在水平方向上拆分。如果要指定在垂直方向上拆分的话，使用参数 JSplitPane.VERTICAL_SPLIT。在创建了拆分窗格以后，也可以通过调用 setOrientation()方法改变拆分方向。

在拆分线上有两个小箭头，用户可以单击以收缩或展开其中任一个窗格。可以通过方法 setOneTouchExpandable()来设置。如果为 true，刚显示小箭头，如上面的代码所示。也可以设置为 false，关闭小箭头。拆分窗格的大小由拆分窗格中的组件的最小的形状大小所决定。在上面的代码中，都设置为宽 100 高 50。在创建了拆分窗格以后，也可以动态地设

置各拆分开的窗格中的组件，使用如下 4 个方法。

```
setLeftComponent
setRightComponent
setTopComponent
setBottomComponent
```

可以在任何时候使用这些方法，而不用考虑拆分窗格的拆分方向。调用 setLeftComponent()和 setTopComponent()是等价的，如果是水平拆分，那么就是设置左边窗格的组件，如果是垂直拆分，就是设置上边窗格的组件。同样，调用 setRightComponent()和 setBotomComponent()也是等价的。调用这些方法时，就会用新的组件代替已经存在的组件。

与其他容器一样，JSplitPane 也支持 add()方法。JSplitPane 将第一个组件添加到左边或上边的窗格中。在往拆分窗格中添加组件时，首先要使用 remove()方法移除已经存在的组件。在使用拆分窗格时，经常需要设置窗格中组件的最小容积，以及拆分窗格或包含的组件的首选大小。如果在扩展拆分窗格时，想保持右边的窗格或下边的窗格大小不变，使用 setResizeWeight()方法。例如：

```
splitPane.setResizeWeight(1.0) ;
```

这个方法的参数在 0.0~1.0 之间。如果设为 0.0，那么在放大拆分窗格时，左边的窗格保持大小不变。如果设为 0.5，那么两个窗格同样地缩放。默认情况下，拆分窗格的大小为首选大小。如果想设为非首选大小，使用 setDividerLocation()方法。例如：

```
splitPane.setDividerLocation(150+splitPane.getInsets().left) ;
```

如果想保持右边的窗格为 150 像素宽，使用下面的代码。

```
splitPane.setDividerLocation(splitPane.getSize().width
                        -splitPane.getInsets().right
                        -splitPane.getDividerSize()
                        -150) ;
```

如果拆分面板已经存在并显示了，那么可以设置拆分线位置为拆分窗格的百分比。例如：

```
splitPane.setDividerLocation(0.25) ;
```

如果想要恢复拆分窗格最初的状态，如重定位拆分线的位置，调用 resetToPreferredSizes()方法。

【例 15.7】使用拆分窗格的完整示例程序。

```
package chapter15;
import javax.swing.JSplitPane;
import javax.swing.JScrollPane;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JFrame;
import java.awt.Dimension;
public class SplitPaneDemo{
    JFrame frame;                                  //声明一个 JFrame 类型变量
    JPanel contentPane,leftPane,rightPane;//声明一个内容面板和两个放置标签的面板
    JScrollPane leftScrollPane,rightScrollPane;    //声明一个滚动窗格
    JSplitPane splitPane;                          //声明一个拆分窗格
    JLabel leftLabel,rightLabel;                   //声明两个标签
    public SplitPaneDemo () {
```



```
            //创建并设置窗口
            frame = new JFrame("SplitPaneDemo ");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            //创建拆分窗格及其中的各种组件
            leftLabel = new JLabel("left");    //创建在两个窗格中显示的标签
            rightLabel = new JLabel("right");
            leftPane = new JPanel();           //创建左右两个窗格中的组件——面板
            rightPane = new JPanel();
            leftPane.add(leftLabel);           //将两个标签分别添加到两个面板中
            rightPane.add(rightLabel);
            Dimension preFerredSize = new Dimension(200,100);//设置两个面板的大小
            leftPane.setPreferredSize(preFerredSize);
            rightPane.setPreferredSize(preFerredSize);
            leftScrollPane = new JScrollPane(leftPane);
                                               //创建滚动窗格,包含前面创建的面板组件
            rightScrollPane = new JScrollPane(rightPane);
            //创建带有两个滚动窗格的拆分窗格
            splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,leftScrollPane,
rightScrollPane);
            splitPane.setOneTouchExpandable(true);  //设置拆分线上的小箭头可用
            splitPane.setDividerLocation(150);      //设置初始拆分线的位置
            //为拆分窗格中的两个滚动窗格提供最小的大小限制
            Dimension minimumSize = new Dimension(100,50) ;
            leftScrollPane.setMinimumSize(minimumSize) ;
            rightScrollPane.setMinimumSize(minimumSize) ;
            contentPane = new JPanel();             //创建内容面板
            contentPane.add(splitPane);             //将拆分窗格添加到内容面板中
            frame.setContentPane(contentPane);      //指定内容面板
        }
        //创建 GUI 界面并显示
        private void createAndShowGUI() {
            //显示窗口
            frame.pack();
            frame.setVisible(true);
        }
        public static void main(String[] args) {
            //为事件分发线程预订一个工作：创建并显示本程序的 GUI
            javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    new SplitPaneDemo().createAndShowGUI();
                }
            });
        }
    }
```

在这个程序中，创建了水平拆分的两个窗格。单击两个窗格中间的箭头，可以收缩其中的一个窗格而扩展另外一个窗格。另外，也可以将光标放在中间的分隔线上，左右拖动，以改变两个窗格的大小。还可以通过拆分窗格的嵌套，创建多个拆分的窗口。

15.2.4　选项卡面板 JTabbedPane

使用 JTabbedPane 类，可以使多个组件（如面板）共享同一个空间。通过选择相应的

选项卡标签，用户可以指定显示哪一个组件，如图 15.15 所示。



图 15.15

要创建一个选项卡面板（JTabbedPane 的实例），首先创建希望在选项卡上显示的组件，然后使用 addTab()方法将组件添加到选项卡面板上。

如图 15.15 所示，一个选项面板上还可以有工具提示和快捷键，并且既可以显示文本也可以显示图像。默认情况下，选项卡位于顶部位置。也可以通过使用 setTabPlacement()方法改变选项卡的位置为左、右、顶或者底部。创建 JTabbedPane 对象并添加组件的代码如下所示。

```
JTabbedPane tabbedPane = new JTabbedPane( );
ImageIcon icon = ……;
JComponent panel1 = ……;
tabbedPane.addTab("Tab 1",icon,panel1,"什么也不做");
```

在 addTab()方法中，有 4 个参数，依次为选项卡标题（标签）、图标、添加的组件及工具提示。其中标签文本和图标可以是 null。addTab()方法有多种形式，但都有标题和要添加的组件这两个参数。

另外一个创建选项卡的方式是使用 insertTab()方法，这个方法可以指定所添加的选项卡的索引号。需要注意的是，addTab()方法不允许指定索引。

要在选项面板中切换选项卡，有三种方式：使用鼠标单击要显示的选项卡、使用键盘上的箭头键和使用快捷键。如果在代码中切换选项卡，使用 setSelectedIndex()和 setSelectedComponent()方法。如果想移除某一个选项卡，使用 JTabbedPane 对象的 remove(i)方法，其中参数 i 为要移除的选项卡的索引号。

【例 15.8】创建使用文本框的示例程序。

```
package chapter15;
import javax.swing.JTabbedPane;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JComponent;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.KeyEvent;
public class TabbedPaneDemo{
    JFrame frame;                              //声明一个 JFrame 类型变量
    JPanel contentPane;                        //声明一个内容面板
    public TabbedPaneDemo() {
       //创建选项卡面板
       JTabbedPane tabbedPane = new JTabbedPane();
```



```
    ImageIcon icon = createImageIcon("images/middle.gif");
    //分别创建 4 个选项卡面板内的组件
    JComponent panel1 = makeTextPanel("面板#1"); //调用自定义方法创建一个面板
    tabbedPane.addTab("Tab 1", icon, panel1,"什么也不做");
                                  //添加选项卡 Tab1 到选项卡面板中
    tabbedPane.setMnemonicAt(0, KeyEvent.VK_1);
                                  //设置快捷键，可用 Alt+0 选择第一个选项卡
    JComponent panel2 = makeTextPanel("面板 #2");
    tabbedPane.addTab("Tab 2", icon, panel2,"还是什么也不做");
    tabbedPane.setMnemonicAt(1, KeyEvent.VK_2);
    JComponent panel3 = makeTextPanel("面板#3");
    tabbedPane.addTab("Tab 3", icon, panel3,"仍然什么也不做");
    tabbedPane.setMnemonicAt(2, KeyEvent.VK_3);
    JComponent panel4 = makeTextPanel("面板#4 (首选大小为 410 × 50).");
    panel4.setPreferredSize(new Dimension(410, 50));
    tabbedPane.addTab("Tab 4", icon, panel4,"还是什么也不做");
    tabbedPane.setMnemonicAt(3, KeyEvent.VK_4);
    //将选项卡面板添加到内容面板中
    contentPane = new JPanel(new GridLayout(1, 1));     //创建内容面板
    contentPane.add(tabbedPane);
    //下面设置可滚动选项卡
    tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
}                                 //自定义方法，用来生成一个自定义面板组件
protected JComponent makeTextPanel(String text) {
    JPanel panel = new JPanel(false);           //创建一个面板对象
    JLabel filler = new JLabel(text);           //创建一个标签对象
    filler.setHorizontalAlignment(JLabel.CENTER);//设置标签中文字居中排列
    panel.setLayout(new GridLayout(1, 1));
                                  //设置面板对象的布局为网格布局，1 行 1 列
    panel.add(filler);                          //将标签添加到面板中
    return panel;                               //返回创建的面板对象
}
//返回一个 ImageIcon，如果文件路径无效，返回 null
protected ImageIcon createImageIcon(String path) {
    java.net.URL imgURL = TabbedPaneDemo.class.getResource(path);
    if (imgURL != null) {
        return new ImageIcon(imgURL);
    } else {
        System.err.println("找不到文件: " + path);
        return null;
    }
}
//创建 GUI 界面并显示
private void createAndShowGUI() {
    //创建并设置窗口
    frame = new JFrame("SplitPaneDemo ");    //创建窗体对象，带有标题
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                            //设置当关闭窗体时，退出程序
    frame.setContentPane(contentPane);       //指定内容面板
    //显示窗口
    frame.pack();
    frame.setVisible(true);
```

```
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作：创建并显示本程序的 GUI
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                UIManager.put("swing.boldMetal", Boolean.FALSE);
                new TabbedPaneDemo().createAndShowGUI();
            }
        });
    }
}
```

在这个程序中，创建了 4 个选项卡面板，每个选项卡都带有图标。每个选项卡都有自定义的布局方式。使用选项卡面板，可以大大地减少空间的占用，可以在有限的空间内显示丰富的信息，并有助于对信息进行清晰的分类。

15.2.5 工具栏 JToolBar

JToolBar 是一个容器组件，用来将许多组件（通常是带图标的按钮）组织到一行或一列。一般来说，工具栏提供与菜单中相对应的便捷访问方式。使用工具栏如图 15.16 所示。



图 15.16　工具栏

要创建一个工具栏（JToolBar 的实例），首先创建希望在工具栏上显示的按钮组件，然后使用 addTab()方法将组件添加到工具栏上。

如图 15.16 所示，一个工具栏上还可以有工具提示和快捷键，并且既可以显示文本也可以显示图像。创建 JToolBar 对象并添加组件的代码如下所示。

```
JToolBar toolBar = new JToolBar("还可以拖动");   //创建工具栏对象
JButton button = ……                              //创建工具栏上的快捷按钮
toolBar.add(button);                              //向工具栏上添加快捷按钮
```

【例 15.9】创建使用工具栏的示例程序。

```
package chapter15;
import javax.swing.JToolBar;
import javax.swing.JButton;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;
import java.net.URL;
import java.awt.BorderLayout;
import java.awt.Dimension;
public class ToolBarDemo{
```



```
        static final private String PREVIOUS = "previous";
        static final private String UP = "up";
        static final private String NEXT = "next";
        static final private String newline = "\n";
        JPanel contentPane;                                    //声明面板对象
        JFrame frame;                                          //声明窗体对象
        public ToolBarDemo() {
            //创建工具栏
            JToolBar toolBar = new JToolBar("还可以拖动");//创建工具栏对象
            addButtons(toolBar);                               //调用 addButtons 方法，
为工具栏添加工具按钮
            //创建文本域和滚动窗格
            JTextArea textArea = new JTextArea(5, 30);
            textArea.setEditable(false);     //将文本域设为不可编辑的
            JScrollPane scrollPane = new JScrollPane(textArea);
                                             //将文本域添加到滚动面板中
            //创建内容面板,布局工具栏和滚动面板
            contentPane = new JPanel(new BorderLayout());
                                             //创建 BorderLayout 布局的面板对象
            contentPane.setPreferredSize(new Dimension(450, 130));
                                             //设置面板的首选大小
            contentPane.add(toolBar, BorderLayout.PAGE_START);
                                             //将工具栏添加到内容面板上面位置
            contentPane.add(scrollPane, BorderLayout.CENTER);
                                             //将滚动面板添加到内容面板中间位置
        }
        //添加工具栏的方法
        protected void addButtons(JToolBar toolBar) {
            JButton button = null;
            //第一个按钮
            button = makeNavigationButton("back", PREVIOUS, "返回到上一步
","Previous");
            toolBar.add(button);
            //第二个按钮
            button = makeNavigationButton("up", UP, "到上一级","Up");
            toolBar.add(button);
            //第三个按钮
            button = makeNavigationButton("forward", NEXT, "前进到下一步","Next");
            toolBar.add(button);
        }
        //创建工具栏按钮的方法，返回创建的工具栏按钮
        protected JButton makeNavigationButton(String imageName,
                                               String actionCommand,
                                               String toolTipText,
                                               String altText) {
            //查找图标
            String imgLocation = "images/" + imageName + ".gif";
            URL imageURL = ToolBarDemo.class.getResource(imgLocation);
            //创建并初始化按钮
            JButton button = new JButton();
            button.setActionCommand(actionCommand);      //设置按钮的动作命令
            button.setToolTipText(toolTipText);          //设置按钮上的提示文本
```

```
            if (imageURL != null) {                          //找到图标
                button.setIcon(new ImageIcon(imageURL, altText));
            } else {                                         //没有找到图标
                button.setText(altText);
                System.err.println("没有找到图标: " + imgLocation);
            }
            return button;
        }
        //创建 GUI 界面并显示
        private void createAndShowGUI() {
            //创建并设置窗口
            frame = new JFrame("ToolBarDemo");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setContentPane(contentPane);               //指定内容面板
            //显示窗口
            frame.pack();
            frame.setVisible(true);
        }
        public static void main(String[] args) {
            //为事件分发线程预订一个工作：创建并显示本程序的 GUI
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                 UIManager.put("swing.boldMetal", Boolean.FALSE);
                 new ToolBarDemo().createAndShowGUI();
                }
            });
        }
    }
```

编译并运行上述程序，可以得到如图 15.16 所示界面。在这个程序中，创建了一个 JToolBar 类型的对象代表工具栏。自定义了两个方法 addButtons() 和 makeNavigationButton()。其中 makeNavigationButton()方法创建工具栏按钮并返回。而 addButtons()方法则用来设置并添加工具栏。

15.3 特殊用途窗口控件

特殊用途窗口控件包括内部窗体组件 JInternalFrame、层面板 JLayeredPane 和根面板 JRootPane。这些特殊用途容器在 UI 中扮演重要的角色。但是对于程序员来说，很少直接使用这些控件。下面就对这些控件做简单介绍。

15.3.1 内部窗体组件 JInternalFrame

使用 JInternalFrame 类，可以在一个窗口中显示另一个类似 JFrame 的窗口。通常，内部窗体组件会被添加到一个桌面面板中（JDesktopPane 对象，是 JLayeredPane 的子类，具有管理重叠的内部窗体的 API）。而相应地，桌面面板被用做 JFrame 的内容面板。图 15.17 显示了一个应用程序的常规的窗体，它具有两个内部窗体。

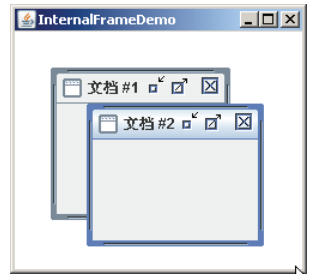


图 15.17 内部窗体



【例 15.10】创建桌面面板和内部窗体的代码如下所示。

```
package chapter15;
import javax.swing.JFrame;
import javax.swing. JInternalFrame;
import javax.swing.JDesktopPane;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;
//这个类自定义了一个内部窗体，它是 JInternalFrame 的子类
class MyInternalFrame extends JInternalFrame {
    static int openFrameCount = 0;
    static final int xOffset = 30, yOffset = 30;
    public MyInternalFrame() {
        super("文档 #" + (++openFrameCount),          //设置内部窗体的标题
              true,       //可调整大小
              true,       //可关闭的
              true,       //可最大化的
              true);  //可最小化的
        this.setSize(300,300);                        //设置内部窗体的大小
        //设置内部窗体的位置
        setLocation(xOffset*openFrameCount, yOffset*openFrameCount);
    }
}
public class InternalFrameDemo{
    JDesktopPane desktopPane;                         //定义桌面面板
    JFrame mainFrame;                                 //定义主窗体
    public InternalFrameDemo() {
        desktopPane = new JDesktopPane();             //创建桌面面板对象
        createFrame();                                //创建第一个内部窗体
        createFrame();                                //创建第二个内部窗体
        desktopPane.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);//指定拖放模式
    }
     //创建一个新的内部窗体
    protected void createFrame() {
        MyInternalFrame frame = new MyInternalFrame(); //创建自定义内部窗体对象
        desktopPane.add(frame);          //将自定义的内部窗体添加到桌面面板中
        frame.moveToFront();             //将当前窗体移动到最前面
    }
    //创建 GUI 界面并显示
    private void createAndShowGUI() {
        //创建并设置窗口
        mainFrame = new JFrame("InternalFrameDemo");    //创建主窗体
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainFrame.setContentPane(desktopPane);          //指定内容面板
        //显示窗口
        mainFrame.setSize(400,410);
        mainFrame.setVisible(true);
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作：创建并显示本程序的 GUI
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
             UIManager.put("swing.boldMetal", Boolean.FALSE);
             new InternalFrameDemo().createAndShowGUI();
            }
```

```
            });
        }
    }
```

编译并运行上述程序，可以得到如图 15.17 所示的界面。使用内部窗体的代码在许多方面与使用常规 Swing 窗体的代码很相似。内部窗体也有根面板，设置 JInternalFrame 的 GUI 与设置 JFrame 的 GUI 非常类似。JInternalFrame 也提供另外的 API，如 pack()，与 JFrame 很类似。内部窗体必须调用 setVisible(true)或 show()方法来显示，否则它不会自动出现。

然而，内部窗体不是窗口或顶层容器。例如，必须将一个内部窗体添加到一个容器中（通常是 JDesktopPane）；一个内部窗体不能是容器层级的根。另外，内部窗体不能产生窗口事件。

另外，内部窗体还具有自己所独有的特性，例如，可以使用 moveToFront()方法将内部窗体移动到屏幕的最上方（不过这只有在内部窗体的容器是层级面板如 JDesktopPane 时才可以使用）。下面是使用内部窗体的原则：

- 必须设置内部窗体的大小。如果不设置内部窗体的大小，那么它将是 0 大小并且永远不会可见。可以使用以下方法之一来设置其大小：setSize()、pack()或 setBounds()。
- 应该设置内部窗体的位置。可以使用 setLocation()或 setBounds()方法来指定内部窗体相对与其容器的左上角坐标。如果不设置的话，默认为 0，0。
- 向内部窗体添加组件时，要将这些组件添加到内部窗体的内容面板中。这与 JFrame 一样。
- 作为内部窗体的对话框，应该使用 JOptionPane 或 JInternalFrame 来实现，而不能使用 JDialog。
- 必须将一个内部窗体添加到一个容器中。通常是 JDesktopPane。如果不这样做的话，内部窗体将不会显现。
- 默认情况下，内部窗体是不可见的。必须调用内部窗体对象的 setVisible(true)或 show()方法来显示内容窗体。
- 内部窗体能产生内部窗体事件，而不是窗口事件。处理内部窗体事件几乎与处理窗口事件等价。

15.3.2 层面板 JLayeredPane

层面板是一个 Swing 容器组件，为定位组件提供第三维：*Z* 轴。当添加一个组件到一个层面板时，使用一个整数指定它的“深度”。数值越高，组件在容器中越接近“顶层”位置。如果组件重叠，上层的组件在低层的组件之上被绘制。

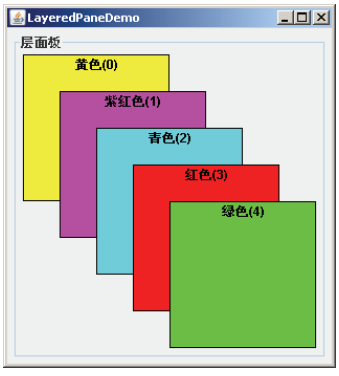


图 15.18　层面板

每一个 Swing 容器都有一个根面板，如 JFrame、JApplet、JDialog 或 JInternalFrame。根面板自动地有一个层面板。大多数程序不会显式地使用根面板的层面板。

Swing 中有两个层面板的类：JLayeredPane 和 JDesktopPane。其中 JDesktopPane 是 JLayeredPane 的子类，它是专用于处理内部窗体任务的。图 15.18 是一个应用程序所创建的层面板，上面有各种重叠的彩色标签，分别在不同的层上。



下面是创建层面板的代码。

```
JLayeredPane layeredPane = new JLayeredPane();
```

层面板只有一个构造器，没有参数。要设置层面板的大小，使用继承过来的setPreferredSize()方法，如以下代码所示。

```
layeredPane.setPreferredSize(new Dimension(300,310));
```

要设置层面板的标题，使用下面的代码：

```
layeredPane.setBorder(BorderFactory.createTitledBorder("层面板");
```

要向层面板中添加组件，使用 add 方法，并指定组件所处的层。在图 15.18 中，一共有 5 个彩色标签。

【例 15.11】创建如图 15.18 所示的层面板。

```
package chapter15;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
public class LayeredPaneDemo{
    //声明在各层上显示的字符串
    private String[] layerStrings = { "黄色(0)", "紫红色(1)", "青色(2)",  "
红色(3)", "绿色(4)" };
    //声明各层的颜色
    private  Color[]  layerColors  =  {  Color.yellow,  Color.magenta,
Color.cyan,Color.red,Color.green };
    private JLayeredPane layeredPane;                    //定义层面板
    private JFrame mainFrame;                            //定义主窗体
    public LayeredPaneDemo() {
        //创建和设置层面板
        layeredPane = new JLayeredPane();
        layeredPane.setPreferredSize(new Dimension(300, 310));
//设置层面板的首选大小
        layeredPane.setBorder(BorderFactory.createTitledBorder("层面板"));
//设置层面板的标题
        //第一个被添加的标签的原点
        Point origin = new Point(10, 20);
        //用于计算下一个标签的偏移量
        int offset = 35;
        //使用绝对位置和大小,添加若干个重叠的彩色标签到层面板中
        for (int i = 0; i < layerStrings.length; i++) {
           //调用自定义方法 createColoredLaybel，创建和设置彩色标签
            JLabel     label     =     createColoredLabel(layerStrings[i],
layerColors[i], origin);
            layeredPane.add(label, new Integer(i));
                                  //第 1 个参数代表要添加的组件，第 2 个参数代表层
            origin.x += offset;                          //改变标签出现的坐标位置
            origin.y += offset;                          //改变标签出现的坐标位置
        }
    }
     //创建和设置彩色标签
    private JLabel createColoredLabel(String text,Color color,Point origin) {
        JLabel label = new JLabel(text);                 //创建标签对象
```

```
        label.setVerticalAlignment(JLabel.TOP);//设置标签上文本排列位于标签顶端
        label.setHorizontalAlignment(JLabel.CENTER);//设置标签上文本水平居中排列
        label.setOpaque(true);                  //设置标签为不透明
        label.setBackground(color);             //设置标签的背景颜色
        label.setForeground(Color.black);       //设置标签上文本的前景色
        label.setBorder(BorderFactory.createLineBorder(Color.black));
                                                //设置标签的边框样式为黑色实边框
        label.setBounds(origin.x, origin.y, 140, 140);//设置标签出现的位置和大小
        return label;                                   //返回设置好的标签对象
    }
    //创建 GUI 界面并显示
    private void createAndShowGUI() {
        //创建并设置窗口
        mainFrame = new JFrame("LayeredPaneDemo");  //创建主窗体
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JComponent contentPane = new JPanel();
        contentPane.setOpaque(true);                //内容面板必须是不透明的
        contentPane.add(layeredPane);               //将层面板添加到内容面板中
        mainFrame.setContentPane(contentPane);      //指定内容面板
        //显示窗口
        mainFrame.pack();
        mainFrame.setVisible(true);
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作：创建并显示本程序的 GUI
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              new LayeredPaneDemo().createAndShowGUI();
            }
        });
    }
}
```

正如图 15.18 所示，添加层时指定的参数越大，层越在上面。还可以通过调用方法 setLayer()动态改变层的层级。例如：

```
layeredPane.setLayer(labelName,3,position);
```

这个方法有三个参数：第 1 个参数是要改变的组件名称，第 2 个参数是一个 int 类型的值，代表新的层，第 3 个参数是在新的层中的位置。在同一个层中，可以使用 moveToFront()和 moveToBack()方法将组件置前或置后。例如：

```
if (…)
       layeredPane.moveToFront(oneLabel);           //将 oneLabel 组件置前
else
       layeredPane.moveToBack(oneLabel);            //将 oneLabel 组件置后
```

15.3.3 根面板 JRootPane

一般情况下，不直接创建 JRootPane 对象。相反，当实例化 JInternalFrame 或一个顶层 Swing 容器时，如 JApplet、JDialog 和 JFrame 时，要获得 JRootPane 对象。

一个根面板由 4 部分组成：



- 透明面板。默认情况下是隐藏的。如果使透明面板可见的话，那么它就像一层透明的玻璃一样覆盖在根面板所有其他部分上面。它可以拦截用于根面板的输入事件。
- 层面板。由内容面板和可选的菜单栏组成。还可以在一个特殊的 *Z* 轴放其他组件。
- 内容面板。它是根面板的可视组件的容器（不包括菜单栏）。
- 可选的菜单栏。如果容器有菜单栏的话，通常使用容器的 setJMenuBar 方法来将菜单栏放置在合适的位置上。

1．透明面板

如果想捕获事件，或者在已经包含组件的区域上绘制，透明面板是很有用的。例如，可以通过使用透明面板截获事件，使用于多组件区域的鼠标事件无效。或者，使用透明面板在多个组件上显示一个图像。例如，设置一个透明面板，使用下面的代码：

```
MyGlassPane myGlassPane = new MyGlassPane(...);//创建一个自定义的透明面板
frame.setGlassPane(myGlassPane);        //将自定义的透明面板设为窗体的透明面板
myGlassPane.setVisible(true);           //显示透明面板
```

可以通过如下的代码获得窗体的透明面板：

```
JComponent glassPane = frame.getGlassPane();
```

2．层面板

每一个根面板将其菜单栏和内容面板放在一个 JLayeredPane 的实例中。层面板的 *Z* 轴顺序使得它可以显示一个弹出菜单在其他组件上。图 15.20 表示层面板的各个功能层及它们之间的关系。

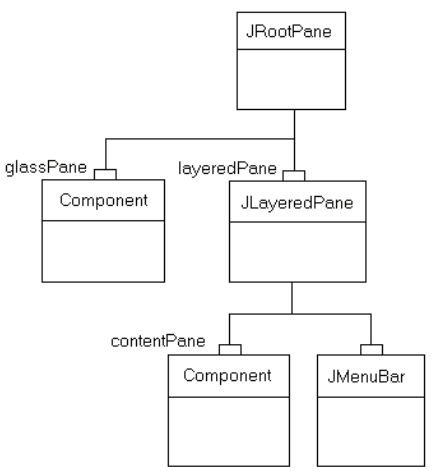


图 15.19　根面板的组成

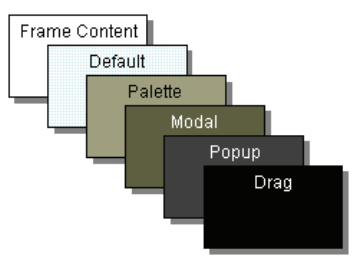


图 15.20　层面板的各个功能层

表 15.1 描述了各个层的功能并列出了与各个层相对应的 JLayeredPane 的常量。

表 15.1　层面板中各个层描述

层名称	值	描述
FRAME_CONTENT_LAYER	new Integer(-30000)	根面板将菜单栏和内容面板添加到层面板的这一层
DEFAULT_LAYER	new Integer(0)	当向层面板添加组件时不指明其所在层，默认放在这个层

续表

层名称	值	描述
PALETTE_LAYER	new Integer(100)	这个层用于浮动工具栏和调色板
MODAL_LAYER	new Integer(200)	模态内部窗体对话框属于这个层
POPUP_LAYER	new Integer(300)	弹出菜单在这个层
DRAG_LAYER	new Integer(400)	当一个组件被拖动时，使用这个层。当拖动时，组件应该返回到它的常规层

15.4 在 Swing 组件中使用 HTML

许多 Swing 组件在它们的 GUI 中显示一个文本字符串。默认情况下，组件的文本以一种单一的字体和颜色显示，并且都显示在一行上。通过调用组件的 setFont()和 setForeground()方法，可以设置组件文本的字体和颜色。例如，下面的代码创建一个标签，然后设置它的字体和颜色：

```
JLabel label = new JLabel("标签 A");
label.setFont(new Font("宋体",Font.PLAIN,14);
label.setForeground(new Color(0xffffdd));
```

如果想在文本中混合字体和颜色，或者想格式化为多行显示，可以使用 HTML。HTML 格式化可以被用于所有的 Swing 按钮、菜单项、标签、工具提示及选项卡面板，以及使用标签来渲染文本的树组件和表组件。

要指定一个组件的文本具有 HTML 格式，只需要将<html>标签放在文本的开始，然后在其余地方使用任何有效的 HTML 即可。下面的代码在一个按钮的文本中使用了 HTML：

```
button = new JButton("<html><b><u>T</u>wo</b><br>lines</html>");
```

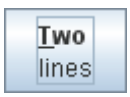


图 15.21 使用 HTML 格式的按钮

按钮的效果如图 15.21 所示。

【例 15.12】在应用程序中的标签上使用 HTML 格式设置字体和颜色。

```
package chapter15;
import javax.swing.*;
import java.awt.*;
public class HtmlDemo{
    JPanel labelPanel;                                    //创建放置标签的面板
    public HtmlDemo() {
        //定义一个初始字符串，带有 HTML 标签
        String initialText = "<html>\n" +
               "HTML 字体和颜色:\n" +
               "<ul>\n" +
               "<li><font color=red>red</font>\n" +
               "<li><font color=blue>blue</font>\n" +
               "<li><font color=green>green</font>\n" +
               "<li><font size=-2>small</font>\n" +
               "<li><font size=+2>large</font>\n" +
               "<li><i>italic</i>\n" +
               "<li><b>bold</b>\n" +
               "</ul>\n";
        //创建一个带有 HTML 格式文本的标签，自定义其大小
        JLabel htmlLabel = new JLabel(initialText) {
```



```
            public Dimension getPreferredSize() {    //定义标签首选大小
                return new Dimension(200, 200);
            }
            public Dimension getMinimumSize() {      //定义标签最小大小
                return new Dimension(200, 200);
            }
            public Dimension getMaximumSize() {      //定义标签最大大小
                return new Dimension(200, 200);
            }
        };
        //设置标签的水平和垂直对齐格式
        htmlLabel.setVerticalAlignment(SwingConstants.CENTER);
                                                     //标签文本垂直居中
        htmlLabel.setHorizontalAlignment(SwingConstants.CENTER);
                                                     //标签文本水平居中
        //创建一个 JPanel 对象，将带有 HTML 格式的标签放置其中
        labelPanel = new JPanel();
        labelPanel.setLayout(new BoxLayout(labelPanel, BoxLayout.PAGE_AXIS));
        labelPanel.setBorder(BorderFactory.createCompoundBorder(
                    BorderFactory.createTitledBorder("使用 HTML 格式的标签"),
                    BorderFactory.createEmptyBorder(10,10,10,10)));
        labelPanel.add(htmlLabel);                   //将 HTML 标签添加到面板中
    }
    //创建 GUI 界面并显示
    private void createAndShowGUI() {
        //创建并设置窗口
        JFrame mainFrame = new JFrame("HtmlDemo");//创建主窗体
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JComponent contentPane = new JPanel();  //创建一个面板组件
        contentPane.add(labelPanel);            //将标签面板添加到内容面板中
        contentPane.setBorder(BorderFactory.createEmptyBorder(10,10,10,
10));                                           //设置边界为 10 像素
        mainFrame.setContentPane(contentPane);  //指定内容面板
        //显示窗口
        mainFrame.pack();
        mainFrame.setVisible(true);
    }
    public static void main(String[] args) {
        //为事件分发线程预订一个工作：创建并显示本程序的 GUI
        SwingUtilities.invokeLater(new
Runnable() {
            public void run() {

    UIManager.put("swing.boldMetal",
Boolean.FALSE);
                    new
HtmlDemo().createAndShowGUI();
            }
        });
    }
}
```



图 15.22　带有 HTML 格式的标签

编译并运行此程序，运行结果如图 15.22 所示。

在这个程序中，创建了一个 JPanel 对象，并将带有 HTML 格式的标签放置其中。在创建显示 HTML 格式的文本字符串时，一定要注意的是，字符串以“<html>”开头，否则 HTML 标记将不能起作用。

15.5 顶层容器对话框 JDialog

一个对话框窗口是一个独立的子窗口，在程序的主窗口这外显示临时信息。为了方便，很多 Swing 组件类能直接实例化并显示对话框。要创建简单、标准的对话框，使用 JOptionPane 类。要创建一个自定义对话框，直接使用 JDialog 类。

15.5.1 对话框简介

每一个对话框都依赖于一个 Frame 组件。当 Frame 被销毁时，所有依赖于它的对话框也都被销毁。当窗体被最小化时，依赖于它的对话框也从屏幕上消失。当窗体被非最小化时，依赖于它的对话框重新出现在屏幕上。在 Swing 中，JDialog 类从 AWT 的 Dialog 类继承了这些行为。

对话框可以是模态对话框。当一个模态对话框可视时，它会阻塞用户对于程序中所有其他窗口的输入。JOptionPane 创建的对话框都是模态的。要创建一个非模态对话框，必须直接使用 JDialog 类。从 JDK1.6 开始，使用新的 Modality API 可以修改对话框窗口的模态形式。

JDialog 类是 AWT 中的 java.awt.Dialog 类的子类。它添加了一个根面板容器，并支持对于 Dialog 对象的默认关闭操作。这些特性与 JFrame 类相同。



图 15.23 简单的消息对话框

甚至当使用 JOptionPane 来实现一个对话框时，实际上仍然在后台使用一个 JDialog。因为 JOptionPane 是一个简单的容器，能自动地创建一个 JDialog 并将其自身添加到 JDialog 的内容面板。创建一个简单对话框的代码量非常少。例如，图 15.23 是一个信息提示对话框。

下面是创建并显示这个对话框的代码：

```
JOptionPane.showMessageDialog(frame,"这是一个简单的消息对话框!");
```

15.5.2 使用 JOptionPane 类创建简单对话框

使用 JOptionPane 类，可以快速创建和定制许多不同类型的对话框。JOptionPane 提供对标准对话框布局的支持，提供相应的提示图标、指定对话框标题和文本，并自定义按钮文本。另外，还可以指定对话框在屏幕的什么位置出现。

JOptionPane 的图标支持，可以让用户很容易指定对话框显示哪种图标。可以使用自定义的图标、没有图标或任何 JOptionPane 的 4 个标准图标之一（问题图标、信息图标、警告图标和错误提示图标）。表 15.2 显示了不同程序外观下这 4 种图标的样式。

表 15.2 JOptionPane 所使用的图标

图标类型	问题图标	信息图标	警告图标	错误提示图标
Java 程序外观				
Windows 程序外观				



对于大多数的简单模态对话框，使用 JOptionPane 类的 showXxxDialog()方法创建和显示。如果是内部窗体对话框，在 show()后面加上 Internal，例如，showMessageDialog()变为 showInternalMessageDialog()。如果想自己控制对话框的关闭行为，或者想让对话框变为非模态的，那么直接初始化一个 JOptionPane 对象并添加到一个 JDialog 实例中，然后调用其 setVisible(true)方法将其显示出来。

两个最有用的 showXxxDialog()方法是 showMessageDialog()方法和 showOptionDialog()方法。其中 showMessageDialog()方法显示一个简单的只带一个按钮的对话框。而 showOptionDialog()方法显示一个自定义的对话框，它可以显示各种带有自定义文本的按钮，能包含一个标准的文本消息或一系列组件。

另外还有两个经常使用的 showXxxDialog()方法是 showConfirmDialog()方法和 showInputDialog()方法。其中方法 showComfirmDialog()又称为确认对话框，通常用来要求用户进行一些确认，但是只提供标准的按钮文本（Yes/No，或者本地化的文本，确定/取消）。而 showInputDialog()方法用来显示一个模态对话框，获得用户的输入字符串。

1．showMessageDialog()方法

调用这个方法会显示一个模态对话框，带有一个按钮。在方法中，可以指定要显示的消息、图标和对话框标题。下面是使用 showMessageDialog()方法的示例代码：

```
JOptionPane.showMessageDialog(frame,"这是一个简单的消息对话框！","消息");
```

上面的代码显示的对话框使用默认的标题和图标。其创建的对话框如图 15.23 所示。下面的代码创建一个带有自定义标题和警告图标的对话框：

```
JOptionPane.showMessageDialog(frame,"这是一个警告信息提示", "警告
",JOptionPane.WARNING_MESSAGE);
```

其创建的对话框如图 15.24 所示。

下面的代码创建一个带有自定义标题和错误提示图标的对话框：

```
JOptionPane.showMessageDialog(frame,"这是一个错误信息提示", "错误提示",
                                    JOptionPane.ERROR_MESSAGE);
```

其创建的对话框如图 15.25 所示。



图 15.24　警告信息对话框



图 15.25　错误信息对话框

下面的代码创建一个带有自定义标题和问题提示图标的对话框。

```
JOptionPane.showMessageDialog(frame,"这是一个问题吗？", "疑问",
                                    JOptionPane.QUESTION_MESSAGE);
```

其创建的对话框如图 15.26 所示。下面的代码创建一个带有自定义标题和不带图标的对话框。

```
JOptionPane.showMessageDialog(frame,"这是一个纯文本对话框。", "不带图标",
                                    JOptionPane.PLAIN_MESSAGE);
```

其创建的对话框如图 15.27 所示。



图 15.26　问题提示对话框



图 15.27　不带图标的对话框

下面的代码创建一个带有自定义标题和自定义图标的对话框。

```
JOptionPane.showMessageDialog(frame,"这是一个自定义的对话框。", "自定义图标",
                                        JOptionPane.INFORMATION_MESSAGE,icon);
```

其创建的对话框如图 15.28 所示。



图 15.28　自定义图标的对话框

2．showOptionDialog()方法

调用这个方法会显示一个模态对话框，带有指定的按钮、图标、消息和标题等。使用这个方法，可以改变出现在按钮上的文本。下面是使用 showOptionDialog()方法的示例代码。

```
    //自定义按钮文本
    Object[ ] options = {"是的，请确定！","不，谢谢！","不好意思，不能确定！"};
    int n = JOptionPane.showOptionDialog(frame,"您的决定\n 请选择：","这仅是一个示例",
                                        JOptionPane.YES_NO_CANCEL_OPTION,
                                                        //有三个按钮
                                        JOptionPane.QUESTION_MESSAGE,
                                                        //有问题图标
                                        null,           //不使用自定义图标
                                        options,        //3 个按钮上的文本
                                        options[2]);    //第 3 个按钮为默认
```

在上面的代码中，第一个参数是对话框的父组件（showXxxDialog()方法都是相同的），它必须是一个窗体、窗体中的一个组件，或者 null。

第二个参数是在对话框的主区域显示的文本内容，在这个字符串中也可以使用转义字符'\n'来换行。

第三个参数为对话框的标题。

第四个参数指定在对话框中出现的按钮，为以下设置之一：DEFAULT_OPTION、YES_NO_OPTION、YES_NO_CANCEL_OPTION、OK_CANCEL_OPTION。

第五个参数指定要在对话框中显示的图标，其值为以下设置之一：PLAIN_MESSAGE（没有图标）、ERROR_MESSAGE、INFORMATION_MESSAGE、WARNING_MESSAGE、QUESTION_MESSAGE。

第六个参数为显示在对话框中的自定义图标。

第七个参数 Object[]指定各个按钮上显示的字符串。第八个参数指定默认被选中的值。

上面的代码所创建的对话框如图 15.29 所示。

3．showConfirmDialog()方法

调用这个方法会显示一个模态对话框，带有两个按钮，让用户做出确认或取消。下面

是使用 showMessageDialog()方法的示例代码。

```
JOptionPane.showConfirmDialog(frame,"请确认信息正确无误.", "请确认",
JOptionPane.YES_NO_OPTION);
```

其创建的对话框如图 15.30 所示。



图 15.29　showOptionDialog 对话框



图 15.30　确认对话框

4．showInputDialog()方法

这是 showXxxDialog()方法中唯一一个不返回整数的方法，而是返回 Object（一般是字符串）。下面是创建一个输入对话框的代码。

```
Object[ ] sexes = {"帅哥", "美女"};
String s = (String)JOptionPane.showInputDialog(
                frame,
                "请选择您的性别：",
                "自定义输入对话框",
                JOptionPane.PLAIN_MESSAGE,
                icon,
                sexes,
                sexes[0]);
```

其创建的对话框如图 15.31 所示。如果不限制用户的输入选择，那么可以将其中的 Object 数组参数改为 null。这时输入框中出现的是一个文本字段，用户可以输入任意的文本，如图 15.32 所示。



图 15.31　选择输入对话框



图 15.32　任意输入对话框

15.6　小结

本章主要讲解 Swing 的高级组件，包括交互式显示高度格式化信息控件，如颜色选择器 JColorChooser、文件选择器 JFileChooser、表组件 JTable 和树组件 JTree 等。另外，还介绍了非顶层容器控件及特殊用途窗口控件。最后两节介绍了如何在 Swing 组件中使用 HTML 以格式化文本，以及 Swing 提供的各种简单而实用的对话框。

读者在学习本章内容时，重点放在对交互式显示格式化信息控件的掌握上，以及各种对话框的使用上。对于非顶层容器控件及特殊用途窗口控件，读者了解即可。

通过本章和上一章的学习，读者基本上对 Swing 中提供的各种容器控件和 Swing 组件有了全面的了解，能熟练使用各种组件。下一章将向读者介绍如何布局程序界面。

第16章 程序界面布局

在应用程序界面上，会有各种各样的组件，有时会很复杂。而界面布局良好的程序，就像一个打扮得体的人一样，看上去美观而友好。在 Java 中，提供了多种称为布局管理器的类来完成界面布局任务，帮助程序员创建高质量的程序界面。

16.1 标准布局管理器简介

很多 AWT 和 Swing 类都提供通用的布局管理器，这些通用的布局管理器是 Java 中的标准布局管理器，包括：

- BorderLayout，边框布局。
- BoxLayout，盒状布局。
- CardLayout，卡片布局。
- FlowLayout，流动布局。
- GridBagLayout，网格包布局。
- GridLayout，网格布局。
- GroupLayout，主要用于集成开发工具使用的布局方式。
- SpringLayout，主要用于集成开发工具使用的布局方式。

16.1.1 BorderLayout 边框布局

每一个内容面板（在所有窗体、applet 和对话框中，内容面板都是主容器）初始化时都是 BorderLayout 布局。在使用 BorderLayout 布局时，组件被放置在 5 个区域：上、下、左、右和中间，如图 16.1 所示。

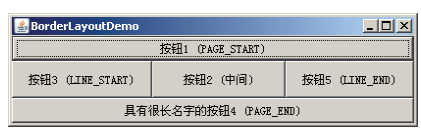


图 16.1 BorderLayout 布局(Windwos 程序外观)

使用 JToolBar 创建的工具栏必须位于一个 BorderLayout 布局的容器中，这样才能将其从开始位置拖放。



16.1.2　BoxLayout 布局

BoxLayout 类将组件放在一个单独的行或列上。它会考虑到组件所要求的最大容积，还允许对组件进行排列，如图 16.2 所示，是将组件放在垂直列上的 BoxLayout 布局。

16.1.3　CardLayout 卡片布局

CardLayout 类可以实现在一个区域中不同的时间包含不同的组件。经常通过一个组合框来控制 CardLayout，由组合框中的状态决定 CardLayout 显示哪一个组件，如图 16.3 所示。



图 16.2　BoxLayout 布局



图 16.3　CardLayout 布局

可以替代使用 CardLayout 的是使用选项卡面板，它提供了相似的功能，但是带有一个预定义的 GUI。CardLayout 布局有时又称为卡片布局。

16.1.4　FlowLayout 流动布局

FlowLayout（流动布局）是 JPanel 类默认的布局方式。它简单地将组件布局在一行上，如果一行放不下的话，就另起一行放置，如图 16.4 所示。



图 16.4　FlowLayout 布局（Windows 程序外观）

16.1.5　GridBagLayout 网格包布局

GridBagLayout（网格包布局）是一个成熟的、具有弹性的布局管理器。它通过将组件放置在网格中的单元格内来排列组件，允许组件跨多个单元格。在网格中的行可以有不同的宽度，网格中的列可以有不同的宽度，如图 16.5 所示。

16.1.6　GridLayout 网格布局

GridLayout（网格布局）将容器按照用户的设置平均划分成若干网格。各个组件依次放入各个单元格中，具有相同的大小，如图 16.6 所示。



图 16.5　GridBagLayout 布局

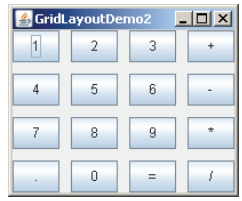


图 16.6　GridBagLayout 布局

16.1.7 GroupLayout 布局

GroupLayout 布局管理器最初被开发的目的是用于 GUI 构建工具，但是它也可被手工使用。GroupLayout 分别对水平和垂直进行布局管理。相应地，每一个组件需要在布局中被定义两次。

16.1.8 SpringLayout 布局

SpringLayout 是一个具有弹性的布局管理器，设计的目的也是用于 GUI 构建器。它可以让程序员指定其控制下组件边缘之间精确地关系。例如，可以定义一个组件左侧与第二个组件右侧一个固定的距离（也可以动态地计算这个距离）。SpringLayout 根据一系列的约束来布局与容器相关联的子组件。

16.2 在程序中使用布局管理器

布局管理器是一个实现了 LayoutManager 接口的对象，它决定一个容器中组件的大小和位置。虽然组件自身可以提供大小和排列方面的设置，但容器的布局管理器有最终决定权。

16.2.1 设置布局管理器

每个 JPanel 对象初始化时都使用 FlowLayout，除非在创建 JPanel 对象时指明不同的布局管理器。内容面板默认情况下使用 BorderLayout 布局。可以任意地改变它们的布局管理器。使用 JPanel 的构造器设置面板对象的布局管理器。例如：

```
JPanel panel = new JPanel(new BorderLayout());
```

在创建容器以后，可以使用 setLayout 方法设置其布局管理器。例如：

```
Container contentPane = frame.getContentPane();
contentPane.setLayout(new FlowLayout());
```

虽然在大多数情况下，都应该使用布局管理器，但实际上也可以不使用。将容器的布局属性设为 null，就不使用布局管理器了。这种情况下，称为“绝对定位”，必须指定容器中每一个组件的大小和位置。缺点是，当调整顶层容器的大小时，布局会显得混乱。绝对定位还不能根据用户和系统的不同而调整。

16.2.2 向容器中添加组件

当向一个 JPanel 对象或内容面板中添加组件时，使用 add()方法。指定给 add 方法的参数依赖于面板或内容面板正在使用的布局管理器。例如，BorderLayout 使用如下的代码，指定组件应该被添加到容器的哪个区域。

```
pane.add(aComponent,BorderLayout.PAGE_START);
```

其中第一个参数是要添加的组件名称，第二个参数是在 BorderLayout 中定义的常量，代表将组件添加到容器的最上边部分。在上一章的许多示例中，已经用到了各种各样的布局管理器及 add 方法中指定的参数。

除 JPanel 和内容面板之外的容器，一般提供有 API 而不使用 add()方法。例如，对于



滚动窗格，并不直接将组件添加到 JScrollPane 中，而是在 JScrollPane 的构造器中指定组件，或使用 setViewportView()方法。因为像这些专用的 API 不需要程序员了解很多 Swing 容器使用哪一种布局管理器。



滚动窗格恰好使用一个名为 ScrollPaneLayout 的布局管理器。

16.2.3　提供组件大小和排列策略

通过指定组件的一个或多个最小、首选和最大尺寸参数，为布局管理器提供自定义的组件大小策略。在上一章讲过，这可以通过调用 setMinimumSize()、setPreferredSize()和 setMaximumSize()方法来设置。也可以创建组件的子类并覆盖合适的 get 方法：getMinimumSize()、getPreferredSize()和 getMaximumSize()。例如，下面的代码将组件的最大容积设置为无限大小：

```
component.setMaximumSize(new Dimension(Integer.MAX_VALUE,Integer.MAX_VALUE);
```

许多布局管理器忽略组件所要求的最大容积设置，除了 BoxLayout 和 SpringLayout。除了提供组件大小容积方面的策略，还可以提供排列的策略。例如，可以指定两个组件的顶端对齐。这可以通过调用组件的 setAlignmentX()和 setAlignmentY()方法，或者通过覆盖组件的 getAlignmentX()和 getAlignmentY()方法，来设置排列策略。

16.2.4　设置组件之间的间隙

有三个因素影响到容器中可视组件之间的间隙：

- 布局管理器。有的布局管理器自动地设置组件之间的间隙，有些则不会。
- 不可视组件。一般在 BoxLayout 布局中使用一些不可视的组件，但它们能占用 GUI 中的空间。
- 空边框。不管什么布局管理器，都可以通过给组件添加空的边框来影响组件之间的空隙的显现。最适合使用空白边框的是没有默认边框的组件，如面板和标签。

16.2.5　设置容器的语言方向

在有的国家和地区，书写语言的顺序是自左向右自上向下，而另外的一些国家和地区，是自右向左，如一些阿拉伯国家。

使用容器的 componentOrientation 属性，可以设置这种语言的方向性。要设置容器语言的方向性，既可以使用 Component 定义的方法 setComponentOrientation()，也可以使用 applyComponentOrientation()在容器的子组件上设置。这两个方法的参数都可以是一个常量，如 ComponentOrientation.RIGHT_TO_LEFT 或 ComponentOrientation.LEFT_TO_RIGHT。例如，下面的代码将容器的语言方向设置为自右向左。

```
Container contentPane = frame.getContentPane();     //获得窗体对象的内容面板
//设置内容面板上的组件方法为自右向左
contentPane. setComponentOrientation(ComponentOrientation.RIGHT_TO_LEFT);
```

也可以调用 ComponentOrientation 的方法 getOrientation(Location)来设置所在地区的语言方向。例如，下面的代码将 JComponent 初始化为阿拉伯语言地区，然后设置内容面板

及它里面所有组件的语言方向。

```
    JComponent.setDefaultLocale(new Locale("ar"));      //设置组件默认的地区
    JFrame frame = new JFrame();
    ...
    Container contentPane = frame.getContentPane();     //获得窗体对象的内容面板
    //对内容面板应用组件排列方向为组件本地排列方向
    contentPane.applyComponentOrientation(ComponentOrientation.getOrientati
on(contentPane.getLocale()));
```

下面的图 16.7 和图 16.8 中，两个程序中的容器的布局管理器都是 FlowLayout，但是设置的语言方向不同，因此组件布局的方向也不同。



图 16.7　默认方向（自左向右）



图 16.8　自由向左的方向

在标准的布局管理器中，支持组件方向的有 FlowLayout、BorderLayout、BoxLayout、GridBagLayout 和 GridLayout。

16.2.6　选择布局管理器

要正确地使用各种布局，首先要了解各布局管理器有各自的优缺点。

- 对于 GridLayout 和 BorderLayout 布局，组件会填满它所在的空间。在使用 BorderLayout 时，对空间要求最大的组件放在中间区域。
- 当要求在一行上按自然大小紧密排列组件时，使用 JPanel 来组织这些组件，并使用 JPanel 的默认布局（FlowLayout）管理器或 BoxLayout 布局管理器。用 SpringLayout 也很好。
- 如果要在行和列上显示同样大小的很多组件，使用 GridLayout 最合适。
- 如果要在一行上或一列上显示一些组件，并且所占的空间大小不同，使用 BoxLayout。
- 如果需要复杂的布局，使用 GridBagLayout，或者使用多个 JPanel 面板分别布局，再组合在一起。
- 另外还有一些第三方的布局管理器，由 Swing 社区开发的，其中最流行的有：TableLayout、MiGLayout 和 Karsten Lentzsch 的 FormLayout。

16.3　使用各种布局管理器

在简单了解了各种类型的布局管理器以后，下面详细介绍如何在程序界面设计中使用这些类型的布局管理器。需要提醒的是，在实际开发中，虽然各种 IDE 集成开发工具提供有很直观的布局管理方式，程序员不需要使用代码来管理布局，但了解各种布局管理器的详细用法和特点，对程序员使用调整界面布局还是有非常大的帮助的。



16.3.1　使用 BorderLayout

正如第 16.1.1 节中的图 16.1 所示，一个 BorderLayout 对象有 5 个区域，这 5 个区域由 BorderLayout 中的常量来指定：

- PAGE_START；
- PAGE_END；
- LINE_START；
- LINE_END；
- CENTER。



在 JDK1.4 之前，这些常量的名称不是这样的，分别由方位名称代表：NORTH、SOUTH、LEFT、RIGHT、CENTER。在使用时，建议使用现在的常量名称，因为它们是标准的名称，并且能使程序根据语言方向的不同而调整。

如果窗口扩大，中间区域会尽可能多地占据空间，而其他 4 个区域只根据需要扩展以填充可用的空间。在很多情况下，容器只会使用到 BorderLayout 对象中的一个或两个区域，如 CENTER，或 CENTER 和 PAGE_END。

在下面的代码中，将组件添加到窗体的内容面板中。默认情况下，内容面板使用 BorderLayout 布局，所以在代码中没有必要再设置布局管理器。

```
//创建并添加第一个按钮
JButton button = new JButton("按钮 1 (PAGE_START)");//创建一个 JButton 按钮对象
pane.add(button, BorderLayout.PAGE_START); //将按钮 button 添加到容器 pane 的顶部
//在 BorderLayout 布局中,中间的组件会显得更大
button = new JButton("按钮 2 (中间)");     //创建第二个 JButton 按钮对象
button.setPreferredSize(new Dimension(200, 100));//设置 button 的首选大小
pane.add(button, BorderLayout.CENTER);     //将第二个按钮添加到容器 pane 的中间区域
button = new JButton("按钮 3 (LINE_START)");     //创建第三个 JButton 按钮对象
pane.add(button, BorderLayout.LINE_START);
                                          //将第三个按钮添加到容器 pane 的左区域
button = new JButton("具有很长名字的按钮 4 (PAGE_END)");
                                             //创建第四个 JButton 按钮对象
pane.add(button, BorderLayout.PAGE_END);    //将第四个按钮添加到容器 pane 的底部
button = new JButton("按钮 5 (LINE_END)");  ////创建第五个 JButton 按钮对象
pane.add(button, BorderLayout.LINE_END);    //将第三个按钮添加到容器pane的右区域
```

在使用 add 方法向 BorderLayout 布局的容器中添加组件时，第一个参数为要添加的组件，第二个参数（使用 BorderLayout 常量）指定将组件添加到的区域。另外，在有的程序中，可能会看到 add 方法的老的用法，将组件作为第二个参数，如下所示。

```
add(BorderLayout.CENTER,button);            //方法有效，但是旧的用法
或者
add("Center",button);                       //方法有效，但是不提倡的用法
```

【例 16.1】实现图 16.1 所示界面布局的程序代码。

```
package chapter16;
import javax.swing.*;
import java.awt.BorderLayout;
```

```
    import java.awt.Container;
    import java.awt.Dimension;
    public class BorderLayoutDemo {
        public static boolean RIGHT_TO_LEFT = false;
        //自定义方法，将组件添加到作为参数传递进来的容器中
        public void addComponentsToPane(Container pane) {
            //判断容器 pane 的布局方式是否是 BorderLayout 布局
            if (!(pane.getLayout() instanceof BorderLayout)) {
                pane.add(new JLabel("容器不使用 BorderLayout 布局!"));
                return;
            }
            //设置组件的自然方向是自右向左
            if (RIGHT_TO_LEFT) {
                pane.setComponentOrientation(java.awt.ComponentOrientation.
RIGHT_TO_LEFT);
            }
            //创建并添加第一个按钮
            JButton button = new JButton("按钮 1 (PAGE_START)");
            pane.add(button, BorderLayout.PAGE_START);//将按钮组件添加到面板上部
            //在 BorderLayout 布局中,中间的组件会显得更大
            button = new JButton("按钮 2 (中间)");
            button.setPreferredSize(new Dimension(200, 100));
                                                    //设置 button 组件的首选大小
            pane.add(button, BorderLayout.CENTER);      //将 button 组件添加到面板中间
            button = new JButton("按钮 3 (LINE_START)");
            pane.add(button, BorderLayout.LINE_START);//将按钮组件添加到面板左侧
            button = new JButton("具有很长名字的按钮 4 (PAGE_END)");
            pane.add(button, BorderLayout.PAGE_END);  //将按钮组件添加到面板底部
            button = new JButton("按钮 5 (LINE_END)");
            pane.add(button, BorderLayout.LINE_END);  //将按钮组件添加到面板右侧
        }
        //创建 GUI 界面并显示. 出于线程安全的目的,这个方法应该从线程分发线程调用
        private void createAndShowGUI() {
            //创建并设置窗体窗口
            JFrame frame = new JFrame("BorderLayoutDemo");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            //设置内容面板
            addComponentsToPane(frame.getContentPane());
            //使用内容面板的默认 BorderLayout 布局.不需要用 setLayout(new BorderLayout());
            //显示窗口
            frame.pack();
            frame.setVisible(true);
        }
        public static void main(String[] args) {
            //使用一个合适的程序外观
            try {
                //使用 windows 程序外观
                UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.
WindowsLookAndFeel");
                //UIManager.setLookAndFeel("javax.swing.plaf.metal.
MetalLookAndFeel");
            } catch (UnsupportedLookAndFeelException ex) {  //捕获并处理异常
```



```
            ex.printStackTrace();
        } catch (IllegalAccessException ex) {
            ex.printStackTrace();
        } catch (InstantiationException ex) {
            ex.printStackTrace();
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        }
        // 关闭粗体字显示
        UIManager.put("swing.boldMetal", Boolean.FALSE);
        //为事件分发线程预订一个工作：创建并显示本程序的 GUI
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new BorderLayoutDemo().createAndShowGUI();
            }
        });
    }
}
```

编译并运行上述程序，可以得到如图 16.1 所示的程序界面。在这个程序中，容器采用了 BorderLayout 布局，将界面划分为 5 个区域。在 BorderLayout 布局下，添加到各个区域的组件会自动地填充满所在的整个区域。

16.3.2　使用 BoxLayout

在 Swing 的包中包括有一个通用目的的布局管理器 BoxLayout。BoxLayout 既可以将组件垂直叠放在一起，也可以将组件排列在一行，就像将一个个的箱子叠放在一起或水平放在一起一样。它与 FlowLayout 相似，但功能上更强大。BoxLayout 布局结果如第 16.1.2 节中的图 16.2 所示。

BoxLayout 的构造器有两个参数，其中第一个参数为它要管理的容器，第二个参数是一个 BoxLayout 类的常量，代表组件布局的方式。如果是 PAGE_AXIS 常量，指定自上向下的组件布局，如果是 LINE_AXIS 常量，指定水平的组件布局（根据容器的 ComponentOrientation 属性确定是自左向右还是自右向左）。例如：

```
JPanel listPane = new JPanel();
listPane.setLayout(new BoxLayout(listPane, BoxLayout.PAGE_AXIS));
                                                    //设置 listPane 容器的布局管理器
JLabel label = new JLabel("标签:");...
listPane.add(label);
listPane.add(Box.createRigidArea(new Dimension(0,5)));
                                                    //添加一个不可见的空白组件
...
//自左向右布局按钮
JPanel buttonPane = new JPanel();
buttonPane.setLayout(new BoxLayout(buttonPane, BoxLayout.LINE_AXIS));
                                                    //设置 listPane 容器的布局管理器
buttonPane.setBorder(BorderFactory.createEmptyBorder(0, 10, 10, 10));
                                                    //设置容器的边框
buttonPane.add(Box.createHorizontalGlue());//添加一个可自动增长的空白组件
buttonPane.add(cancelButton);
buttonPane.add(Box.createRigidArea(new Dimension(10, 0)));
buttonPane.add(setButton);
```

Box 类的静态方法 createRigidArea()用来创建一个不可见的组件，这个不可见的组件通常放在其他组件之间，用来使用空白区域分隔组件。Box 类的静态方法 createHorizontalGlue()创建一个可自动缩放的空白组件，当容器改变时，它会自动增长以占据额外的空间。

在垂直方向上布局组件时，BoxLayout 会调整组件的大小（放大或收缩），以使各个组件的高度之和与容器的高度相匹配。但不会超过各个组件的最小或最大的高度。超出部分的空白区域会出现在容器的底部。容器的宽度自动为其中最宽的组件的宽度。在水平方向上布局组件时，与垂直方向上相似。

由 BoxLayout 布局管理器控制的相邻的组件是紧挨着的。如果想在组件之间有空隙，既可以对其中一个组件或两个组件都添加空边框，或者插入一个不可见的组件。使用 Box 类来创建不可见的组件。

Box 类中定义了一个内嵌类 Box.Filler，这是一个透明的组件，什么也没有绘制，用于在两个组件之间提供空隙。实际上，Filler 并不是真的不可见，只是 setVisible(true)没有被调用。Box 类提供有很简便的方法来帮助创建普通类型的填充物。表 16.1 中对使用 Box 和 Box.Filler 创建不可见组件做了说明。

表 16.1　创建不可见组件

方　法	说　明
Box.createRigidArea(size)	创建一个总是具有指定大小的不可见组件
Box.createHorizontalGlue()	创建一个横向的可自动缩放的 glue 组件
Box.createVerticalGlue()	创建一个纵向的可自动缩放的 glue 组件

Box.createRigidArea(size)方法创建一个指定大小的不可见组件，这可用于设置两个组件间固定的间隙。例如，想在两个两组件间固定 5 个像素的间隙，可以使用如下的代码。

```
container.add(firstComponent);
container.add(Box.createRigidArea(new Dimension(5.0));
container.add(secondComponent);
```

不带间隙的两个组件布局和带 5 个像素间隙的组件布局如图 16.9 和图 16.10 所示。

使用 Box.createHorizontalGlue()和 Box.createVerticalGlue()方法，可以在水平方向上或垂直方向上组件之间增加可自动增长的弹性间隙。例如，对水平方向上的两个按钮之间添加一个 glue 组件，那么显示结果如图 16.11 和图 16.12 所示。



图 16.9　没有固定间隙



图 16.10　有 5 像素宽的固定间隙



图 16.11　没有弹性间隙



图 16.12　有弹性间隙

16.3.3　使用 CardLayout

CardLayout 类管理共享同一显示区域的多个组件（通常是 JPanel 对象实例）。CardLayout 布局管理器所管理的组件就像一副扑克牌或一叠叠放的卡片，在任何时候只有最顶上的卡片才可见。

在第 16.1.3 节中的图 16.3 是一个使用 CardLayout 布局组件的界面。对一个容器面板应用 CardLayout，使用如下的代码。

```
JPanel cards = new JPanel(new CardLayout());
```

CardLayout 类的构造器非常简单，不带参数。将两个面板对象添加到 CardLayout 布局管理器管理的 cards 中，使用 add()方法。

```
JPanel card1 = new JPanel();
JPanel card2 = new JPanel();
……
cards.add(card1,"卡片 1");
cards.add(card2."卡片 2");
```

在 add 方法中的第二个参数，代表添加到容器中的卡片组件的名称，它是用来标识该卡片的。当布局管理器要显示某一个卡片组件时，可以使用此名称，如以下代码所示。

```
CardLayout cl = (CardLayout)(cards.getLayout());
cl.show(cards,"卡片 1");
```

CardLayout 布局管理器调用其 show 显示其所管理的 cards 容器中指定名称（第二个参数）的卡片组件。

16.3.4　使用 FlowLayout

FlowLayout 布局管理器将组件按首选大小排列在一行上。如果一行上放不下，FlowLayout 会自动换行，在多行上排列。如果一行的宽度大小组件所需的宽度，默认情况下组件都居中显示。可以在使用 FlowLayout 构造器构造布局对象是，使用常量参数指定组件对齐方式（靠左、靠右还是居中对齐等）。FlowLayout 共定义了 5 个常量用来指明组件对齐方式。

- FlowLayout.LEADING：与开始一边对齐。
- FlowLayout.TRAILING：与结束一边对齐。
- FlowLayout.CENTER：居中对齐。
- FlowLayout.LEFT：左对齐。
- FlowLayout.RIGHT：右对齐。

例如，图 16.13 所示为与结束边对齐的流动布局。

【例 16.2】实现如图 16.13 所示的程序界面布局。



图 16.13　流动布局，靠结束边对齐

```
package chapter16;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
public class FlowLayoutDemo2{
    FlowLayout flowLayout = new FlowLayout();
    //添加组件到容器中的方法
    public void addComponentsToPane(final Container pane) {
        final JPanel buttonPane = new JPanel();
        buttonPane.setLayout(flowLayout); //设置为流动布局(FlowLayout 布局)
        flowLayout.setHgap(20);          //设置水平方向上组件之间的间距为20像素
        //buttonPane.setAlignment(FlowLayout.LEADING);  //设置与开始一边对齐
        buttonPane.setAlignment(FlowLayout.TRAILING);   //设置与结束一边对齐
        //buttonPane.setAlignment(FlowLayout.LEFT);     //设置左对齐
        //buttonPane.setAlignment(FlowLayout.RIGHT);    //设置右对齐
        //buttonPane.setAlignment(FlowLayout.CENTER);   //设置居中对齐
```

```
            //添加按钮到 buttonPane 中
            buttonPane.add(new JButton("按钮 1"));
            buttonPane.add(new JButton("按钮 2"));
            buttonPane.add(new JButton("按钮 3"));
            buttonPane.add(new JButton("具有很长名字的按钮 4"));
            buttonPane.add(new JButton("5"));
            pane.add(buttonPane, BorderLayout.CENTER);
                                                    //将添加有按钮的面板添加到内容面板中
        }
        // 创建 GUI 界面并显示
        private void createAndShowGUI() {
            //创建并设置窗口
            JFrame frame = new JFrame("FlowLayoutDemo2");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            //设置内容面板
            addComponentsToPane(frame.getContentPane());
            //显示窗口
            frame.pack();
            frame.setVisible(true);
        }
        public static void main(String[] args) {
            // 应用 Windows 程序外观
            try {
                UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.
WindowsLookAndFeel");
            } catch (Exception ex) { }
            javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    new FlowLayoutDemo2().createAndShowGUI();
                }
            });
        }
    }
```

上面的程序中，先创建一个单独的方法 addComponentsToPane()，将内容面板作为参数传递进来。注意，组件对象传递的实际上是引用，所以在此方法中，将组件添加到流动布局的 buttonPane 中以后，再将 buttonPane 面板添加到内容面板中。编译并运行上述程序，可以得到如图 16.13 所示的界面。

16.3.5 使用 GridLayout

由 GridLayout 类实现的布局管理器称为网格布局管理器。GridLayout 的布局方式是将容器按用户的设置平均划分成若干个网格，其中的每一格称为一个“单元格”，如图 16.14 所示。

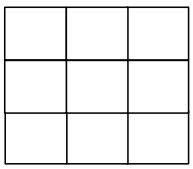


图 16.14 GridLayout 布局方式

在通过构造方法 GridLayout(int row,int cols)创建网格布局管理器对象时,参数 rows 用来设置网格的行数，参数 cols 用来设置网格的列数。在设置时分为以下 4 种情况：

只设置了网格的行数，即 rows 大于 0，cols 等于 0。在这种情况下，容器将先按行排列组件，当组件个数大于 rows 时，则再增加一列，依此类推。



只设置了网格的列数，即 rows 等于 0，cols 大于 0。在这种情况下，容器将先按列排列组件，当组件个数大于 colss 时，则再增加一行，依此类推。

同时设置了网格的行数和列数，即 rows 大于 0，cols 大于 0。在这种情况下，容器将先按行排列组件，当组件个数大于 rows 时，则再增加一列，依此类推。

同时设置了网格的行数和列数，但是容器中的组件个数大于网格数（rows*cols）。在这种情况下，容器将先按行排列组件，当组件个数大于 rows 时，则再增加一列，依此类推。

GridLayout 布局管理器默认组件的水平间距和垂直间距均为 0 像素，可以通过 GridLayout 类的方法 setHgap(int hgap)和 setVgap(int vgap)设置组件的水平间距和垂直间距。

【例 16.3】创建如图 16.6 所示的计算器的界面布局。

```
package chapter16;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class GridLayoutDemo2{
   private String[][] names = {
                       {"1", "2", "3", "+"},
                       {"4", "5", "6", "-"},
                       {"7", "8", "9", "*"},
                       {".", "0", "=", "/"}};
   private JButton[][] buttons = new JButton[4][4];//创建显示界面的数组按钮
   GridLayout experimentLayout = new GridLayout(4,4);
                                       //创建一个 GridLayout 布局管理器的界面
   public GridLayoutDemo2() {
        experimentLayout.setHgap(10);   //设置单元格水平间距为 10 像素
        experimentLayout.setVgap(10);   //设置单元格垂直间距为 10 像素
   }
   //将组件添加到容器面板中的方法
   public void addComponentsToPane(final Container pane) {
       final JPanel compsToExperiment = new JPanel();
                                       //创建添加按钮组件的面板对象
       compsToExperiment.setLayout(experimentLayout);  //设置布局管理器
       //添加按钮到网格单元中
       for(int row=0;row<names.length;row++){
            for(int col=0;col<names.length;col++){
                compsToExperiment.add(new JButton(names[row][col]));
                                       //创建按钮对象，并添加到面板中
            }
       }
       pane.add(compsToExperiment, BorderLayout.NORTH);
   }
   // 创建 GUI 界面并显示
       private void createAndShowGUI() {
       //创建并设置窗口
       JFrame frame = new JFrame("GridLayoutDemo2");
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       //设置内容面板
       addComponentsToPane(frame.getContentPane());
       //显示窗口
       frame.pack();
       frame.setVisible(true);
   }
   public static void main(String[] args) {
```

```
            // 设定程序外观风格
            try {
                //UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.
WindowsLookAndFeel");

UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
                                                        //应用跨平台的外观
            } catch (Exception ex) { }
            // 关闭组体字显示
            UIManager.put("swing.boldMetal", Boolean.FALSE);
            //启动主程序线程
            javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    new GridLayoutDemo2().createAndShowGUI();
                }
            });
        }
    }
```

编译并运行上述程序，可以得到如图 16.6 所示的计算器的界面布局。在上面的程序中，容器采用 GridLayout 布局方式，将界面划分为 4 行 4 列。然后通过两个 for 循环，依次将各个按钮组件添加到单元格中。在这个程序中，设置单元格的水平和垂直间距都为 10 像素，使组件之间有一定的间隔，看上去更加美观。

16.3.6 使用 GridbagLayout

GridbagLayout 又叫网格包布局管理器，它是网格布局管理器的一个扩展，是最灵活又是最复杂的布局管理器之一。在网格包布局中，组件在网格中可以占据多个单元，而且不同的行和列的比例也不必相等。在网格单元内的组件可以用不同的方式进行摆放。



图 16.15　GridBagLayout 布局

本质上，GridBagLayout 将组件放在单元格中，然后使用组件的自然大小来决定单元格应该有多大，如图 16.15 所示。

从图 16.15 中可以看出，这个网格包有三行三列。其中第二行的按钮 4 跨三列单元格；第三行的按钮 5 跨右边的两列单元格。

在网格包布局方式中，指定组件的大小和位置的方式是为每一个组件指定“约束”。给组件设置约束的首先途径是使用容器的 add 方法的变体，并传递给它一个 GridBagConstraints 对象参数。

为了创建一个网格包布局，要使用类 GridBagLayout 和一个叫做 GridBagConstraints 的辅助类。GridBagLayout 是布局管理器，GridBagConstraints 是用来定义放置到单元中的每个组件的属性的（包括位置、大小和对齐方式等）。网格包、约束限制和每个组件之间的关系定义了整个布局的具体情况。一般情况下，创建一个网格包的步骤如下所示。

（1）创建一个 GridBagLayout 对象并将它定义成当前的布局管理器。

（2）创建 GridBagconstraints 的一个新实例。

（3）为某个组件设置该约束限制。

（4）将组件加入到容器中，并应用已设置的约束条件。

在使用 GridBagLayout 时，必须为每一个要加入到容器内的组件设置所有约束信息。



在使用 add 方法将组件添加到 GridBagLayout 布局管理器所管理的容器中时，指定组件要应用的约束限制。例如：

```
JPanel pane = new JPanel(new GridBagLayout());
GridBagConstraints c = new GridBagConstraints();
//对于要添加到容器中的每一个组件：
//...创建该组件...
//...设置 GridBagConstraints 对象中的各个实例变量...
pane.add(theComponent, c); //将组件 theComponent 添加到容器 pane 中，并应用约束限制 c
```

另外，如上面的代码所示，一旦创建并设置了约束 GridBagConstraints 的实例，就可以将该约束应用于多个组件。但是建议不要重用约束对象 GridBagConstraints，因为如果忘了为每一个新的实例对象重置属性时，很容易导致不易察觉的 bug。GridBagConstraints 类有以下的实例变量，用来代表对组件的各项约束：

- gridx，gridy。代表组件在网格包中的坐标，既组件所在的单元格的列和行，最左边的列的 gridx 为 0，最上边的行的 gridy 为 0。如果组件跨多个单元格时，这两个变量代表组件左上角所在的单元格。

例如最左上角的单元格的 gridx 的值为 0，gridy 的值为 0。使用默认值 GridBagConstraints.RELATIVE 指定组件位于上一个被放入的组件的右边（gridx=GridBagConstraints.RELATIVE）或下边（gridy=GridBagConstraints.RELATIVE）。建议明确指定 gridx 和 gridy 的值而不使用 GridBagConstraints.RELATIVE。

- gridwidth，gridheight。分别用来指定组件所占的行或列的数量。这两个约束指定的是组件所使用的单元格数量，而不是它所使用的像素数。默认值是 1，即组件只占一个单元格。

当为 gridwidth 赋予常量 GridBagConstraints.REMAINDER 值时，指定组件位于所在行的最后一行；当为 gridheight 赋予常量 GridBagConstraints.REMAINDER 值时，指定组件位于所在列的最后一列。当为 gridwidth 和 gridheight 赋予常量 GridBageConstraints.RELATIVE 常量值时，指定组件位于所在行或所在列的最后一个组件的后面。

建议明确指定 gridwidth 和 gridheight 的值而不使用常量 GridBagConstraints.REMAINDER 和 GridBagConstraints.RELATIVE。



使用布局管理器 GridBagLayout 时，只有组件位于最左边的一列时，或明确指定组件的 gridx 和 gridy 的值时，才能跨多行。

- fill。当组件所在显示区域的大小超过组件的自然大小时，使用 fill 来决定是否调整组件的大小及如何调整组件的大小。

可以给 fill 变量赋的值均为 GridBagConstraints 中定义的常量，包括默认值 GridBagConstraints.NONE（组件在水平和垂直方向上都保持不变）、GridBagConstraints.HORIZONTAL（组件在水平方向上填充整个显示区域，但在垂直方向上不变）、GridBagConstraints.VERTICAL（组件在垂直方向上填充整个显示区域，但在宽度上不改变）和 GridBagConstraints.BOTH（组件在水平方向和垂直方向上都填满整个显示区域）。

- ipadx，ipady。指定组件的内部填充值，它们决定组件的大小增加多少。默认值为 0。组件的宽度至少是其最小宽度再加上 ipadx*2 像素宽，因为水平和垂直填充都

应用到组件的两个边上。同样，组件的高度至少是其最小高度加上 ipady*2 像素高。

- insets。指定组件的外部填充值，它代表组件及其显示区域之间的最小间距。insets 变量的值是一个 Insets 对象。默认情况下，每一个组件都没有外部填充值。
- anchor。当组件的大小比其所在显示区域小时，使用 anchor 变量来决定在显示区域的什么地方定位组件。可以给 anchor 变量赋予的有效值均为 GridBagConstraints 中定义的常量，包括默认值 GridBagConstraints.CENTER（组件居中）、GridBagConstraints.PAGE_START、GridBagConstraints.PAGE_END、GridBagConstraints.LINE_START、GridBagConstraints.LINE_END、GridBagConstraints.FIRST_LINE_START、GridBagConstraints.FIRST_LINE_END、GridBagConstraints.LAST_LINE_START、GridBagConstraints.LAST_LINE_END。



在 1.4 之前这些常量值是用方位名称代表的，例如 GridBagConstraints.NORTHEAST，代表显示区域的右上部分。但是建议使用新的常量，这样能使程序更容易地本地化。

图 16.16 中显示了各个常量值所代表的位置。

- weightx，weighty。这两个变量的值决定如何在列和行中分配空间。这对指定组件调整大小时的行为有很重要的作用。可以把它们看做代表在宽度上和高度上的“权重”值。

FIRST_LINE_START	PAGE_START	FIRST_LINE_END
LINE_START	CENTER	LINE_END
LAST_LINE_START	PAGE_END	LAST_LINE_END

图 16.16　anchor 的常量值代表的位置

指定权重值是绝对称得上是一项艺术，对于 GridBageLayout 布局管理器控制的组件的表现形式有很重要的作用。如果不为 weightx 和 weighty 至少指定一个非零值的话，所有的组件都会聚集在容器的中间。因为当这两个变量的默认值都是 0.0 时，GridBagLayout 将任何额外的空间都移动到单元格和容器边界之间。

weightx 和 weighty 的值都位于 0.0 和 1.0 之间。值越大，说明组件的行或列应该获得更大的空间。为于每一列来说，权重是相对于指定给该列中的组件最高的 weightx，同时每一个跨多列的组件的权重被在组件所在列之间拆分。相似地，每一行的权重是相对于指定给该行中的组件最高的 weighty。剩余的空间分配给最右边的一列和最底下的一行。

例如，要布局图 16.15 所代表的应用程序的界面，按下面的步骤进行：

① 创建一个 GridBagLayout 布局管理器。

```
pane.setLayout(new GridBagLayout());
                                    //将容器 pane 的布局管理器设置为 GridBagLayout
```

② 创建一个 GridBagConstraints 约束对象。

```
GridBagConstraints c = new GridBagConstraints();
```

③ 为约束对象 c 的变量 fill 赋值，指定当组件的大小小于其所在的显示区域时，应该如何去调整。

```
if (shouldFill) {                              //如果条件 shouldFill 为 true 时
    c.fill = GridBagConstraints.HORIZONTAL;//自然高度，水平方向上填满所在区域
}
```



④ 设置约束对象 c 的 weightx 的值。

```
if (shouldWeightX) {                    //如果条件 shouldWeightX 为 true 时
c.weightx = 0.5;                        //列权重设为 0.5
}
```

⑤ 设置约束对象 c 的 gridx 和 gridy 变量的值，指明组件所处的单元坐标。

```
c.gridx = 0;                            //第 1 列
c.gridy = 0;                            //第 1 行
```

⑥ 创建布局管理器 GridBagLayout 所管理的按钮组件 button。

```
JButton button = new JButton("按钮 1");
```

⑦ 将按钮 button 添加到容器 pane 当中，并应用布局约束对象 c。

```
pane.add(button, c);
```

⑧ 创建第二个按钮组件，并重置约束对象，添加第二个按钮到容器 pane 中并应用布局约束对象。

```
button = new JButton("按钮 2");
c.fill = GridBagConstraints.HORIZONTAL;
c.weightx = 0.5;
c.gridx = 1;
c.gridy = 0;
pane.add(button, c);
```

⑨ 创建第三个按钮组件，并重置约束对象，添加第三个按钮到容器 pane 中并应用布局约束对象。

```
button = new JButton("按钮 3");
c.fill = GridBagConstraints.HORIZONTAL;
c.weightx = 0.5;
c.gridx = 2;
c.gridy = 0;
pane.add(button, c);
```

⑩ 创建第四个按钮组件，并重置约束对象，添加第四个按钮到容器 pane 中并应用布局约束对象。

```
button = new JButton("具有很长名字的按钮 4");
c.fill = GridBagConstraints.HORIZONTAL;
c.ipady = 40;                           //设置内部填充值，使这一行比别的行要高
c.weightx = 0.0;
c.gridwidth = 3;                        //指定组件跨三列显示区域
c.gridx = 0;
c.gridy = 1;
pane.add(button, c);
```

⑪ 创建第五个按钮组件，并重置约束对象，添加第五个按钮到容器 pane 中并应用布局约束对象。

```
button = new JButton("5");
c.fill = GridBagConstraints.HORIZONTAL;
c.ipady = 0;                            //重置为默认值
c.weighty = 1.0;                        //要求任何额外的垂直空间
c.anchor = GridBagConstraints.PAGE_END;//设置组件位于所在显示区域的底部
c.insets = new Insets(10,0,0,0);//设置组件四周的边距，分别为上、左、下、右边距
```

```
c.gridx = 1;                          //与按钮 2 对齐
c.gridwidth = 2;                      //指定该组件跨两列显示区域
c.gridy = 2;                          //第 3 列
pane.add(button, c);
```

【例 16.4】创建如图 16.15 所示的应用程序的界面。

```
package chapter16;
import java.awt.*;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.UIManager;
public class GridBagLayoutDemo {
    final static boolean shouldFill = true;
                                  //定义静态变量 shouldFill，代表组件是否水平延伸
    final static boolean shouldWeightX = true;
                                  //定义静态变量 shouldWeightX，代表权重
    final static boolean RIGHT_TO_LEFT = false;  //设置组件的本地化排列方向
    //自定义方法，将组件添加到内容面板中
    public void addComponentsToPane(Container pane) {
        if (RIGHT_TO_LEFT) {
            pane.setComponentOrientation(ComponentOrientation.RIGHT_TO_LEFT);
        }
        JButton button;
        pane.setLayout(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints();
        if (shouldFill) {
             c.fill = GridBagConstraints.HORIZONTAL;
        }
        //为按钮 1 设置布局约束并应用该布局约束
        button = new JButton("按钮 1");
        if (shouldWeightX) {
             c.weightx = 0.5;
        }
        c.fill = GridBagConstraints.HORIZONTAL;
        c.gridx = 0;
        c.gridy = 0;
        pane.add(button, c);
        //为按钮 2 设置布局约束并应用该布局约束
        button = new JButton("按钮 2");
        c.fill = GridBagConstraints.HORIZONTAL;
        c.weightx = 0.5;
        c.gridx = 1;
        c.gridy = 0;
        pane.add(button, c);
        //为按钮 3 设置布局约束并应用该布局约束
        button = new JButton("按钮 3");
        c.fill = GridBagConstraints.HORIZONTAL;
        c.weightx = 0.5;
        c.gridx = 2;
        c.gridy = 0;
        pane.add(button, c);
        //为按钮 4 设置布局约束并应用该布局约束
        button = new JButton("具有很长名字的按钮 4");
        c.fill = GridBagConstraints.HORIZONTAL;
        c.ipady = 40;
        c.weightx = 0.0;
```



```
        c.gridwidth = 3;
        c.gridx = 0;
        c.gridy = 1;
        pane.add(button, c);
        //为按钮 5 设置布局约束并应用该布局约束
        button = new JButton("5");
        c.fill = GridBagConstraints.HORIZONTAL;
        c.ipady = 0;
        c.weighty = 1.0;
        c.anchor = GridBagConstraints.PAGE_END;
        c.insets = new Insets(10,0,0,0);
        c.gridx = 1;
        c.gridwidth = 2;
        c.gridy = 2;
        pane.add(button, c);
    }
    // 创建并显示 GUI 界面
    private void createAndShowGUI() {
        //创建并设置窗口
        JFrame frame = new JFrame("GridBagLayoutDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //设置内容面板
        addComponentsToPane(frame.getContentPane());
        //显示窗口
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                /* 关闭粗体字显示 */
                UIManager.put("swing.boldMetal", Boolean.FALSE);
                new GridBagLayoutDemo().createAndShowGUI();
            }
        });
    }
}
```

在上面的示例代码中，为所有的组件使用了同一个约束对象 GridBagConstraints，但是在实际开发过程中，建议不要这样重用一个 GridBagConstraints 对象。另外要注意的时，在设置第 4 个按钮时，设置了 ipady 变量值为 40，那么在设置下一个组件的约束时，要将其值重置为默认值：c.ipady=0。

在有的程序中，可能会看到，为组件添加约束限制时，不使用 add()方法，而是使用 setConstraints()方法，如下代码所示。

```
GridBagLayout gridbag = new GridBagLayout();
pane.setLayout(gridbag);
...
gridbag.setConstraints(button, c);    //通过方法 setConstraints 为组件设置约束对象
pane.add(button);                     //添加组件
```

不过建议还是使用 Container.add()方法，这比使用 setConstraints()方法更清晰。网格包布局是最灵活和复杂的布局方式之一，读者可以将示例 16.4 中的各个约束变量的值加以改变，拖动窗体的边框以改变界面的大小，仔细观察不同的约束值对界面布局和组件排列的影响，对其有更深刻的理解。

16.4 实例：设计用户登录界面



图 16.17 用户登录界面

在实际开发中，界面布局是相对比较复杂而细致的一项工作，一般都会借助于各种第三方提供的集成开发工具来进行界面布局。作为练习，本节将综合使用本章所学的界面布局知识，手工设计一个常见的用户登录界面，如图 16.17 所示。

16.4.1 设计界面布局

在实际设计应用程序界面时，往往多种布局管理器组合使用，以达到最佳的布局效果。本实例也不例外，如图 16.18 所示，用户登录界面由两个面板（JPanel 对象）组成，上面的为 topPane，下面的为 bottomPane，这两个面板位于内容面板（内容窗格）中，内容面板默认为 BorderLayout 布局。其中 topPane 位于内容面板的 CENTER 位置，而 bottomPane 位于内容面板的 PAGE_END 位置。

通过 setLayout()方法将 bottomPane 的布局管理器设为 FlowLayout（流动布局），将 topPane 的布局管理器设为 GridBagLayout（网格包布局），如图 16.19 所示。

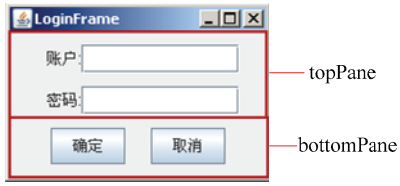


图 16.18 界面由两个 JPanel 对象组成



图 16.19 设计网格

其中，“账户”标签所在的单元格坐标为（0，0），账户文本框所在的单元格坐标为（1，0），“密码”标签所在的单元格坐标为（0，1），密码文本框所在的单元格坐标为（1，1）。需要注意的是，这里的单元格坐标与像素坐标不同，它的坐标值是由（gridx，gridy）组成的，其中最左边的单元的 gridx 值为 0，最上边的单元格的 gridy 为 0。所以账户文本框的单元坐标是（1，0）而不是（0，1）。了解了用户登录界面的布局以后，下面用代码来实现。

16.4.2 代码实现

首先，为 topPane 面板和 bottomPane 设置布局管理器。topPane 为 GridBagLayout 布局，bottomPane 为 FlowLayout 布局。代码如下。

```
JPanel topPane = new JPanel();   //放置标签和文本框的面板
topPane.setLayout(new GridBagLayout());
                              //将topPane面板布局设为网格包布局(GridBagLayout)
JPanel bottomPane = new JPanel();//放置"确定"和"取消"按钮的面板
FlowLayout flowLayout = new FlowLayout();  //创建 FlowLayout 布局管理器对象
flowLayout.setHgap(20);                    //设置流动布局中组件之间的水平间距
flowLayout.setVgap(10);                    //设置流动布局中组件之间的垂直间距
bottomPane.setLayout(flowLayout);//将 flowLayout 设为 bottomPane 的布局管理器
```

在 topPane 面板中，共有 4 个组件，分别为标签“账户”、用户输入账户的文本框、标



签“密码”和用户输入密码的密码框。分别为这 4 个组件创建布局约束对象（即 GridBagConstraints 实例），并分别设置相应的约束对象变量。对于标签“账户”，位于网格布局的单元格（0，0）位置处，靠右对齐，布局代码如下所示。

```
//标签"账户"
GridBagConstraints conLabelName = new GridBagConstraints();
                                                    //创建约束对象
conLabelName.fill = GridBagConstraints.NONE;        //组件保持自然大小
labelName = new JLabel("账户:");        //创建“账户”标签
conLabelName.weightx = 0.2;             //设置“账户”标签的水平方向上权重
conLabelName.gridx = 0;                 //标签所在列为第 1 列
conLabelName.gridy = 0;                 //标签所在行为第 1 行
conLabelName.anchor = GridBagConstraints.LINE_END; //标签在单元格内靠右对齐
topPane.add(labelName, conLabelName);  //将标签加入 topPane 面板中，并应用约束
```

对于用户输入账户的文本框，位于网格布局的单元格（1，0）位置处，布局代码如下所示。

```
//账户文本框
GridBagConstraints conTextFieldName = new GridBagConstraints();
                                        //创建约束对象
conTextFieldName.fill = GridBagConstraints.HORIZONTAL;
                                        //文本框水平扩展整个单元格
textFieldName = new JTextField();
conTextFieldName.weightx = 0.8;         //设置文本框的水平方向上的权重
conTextFieldName.weighty = 0.5;
conTextFieldName.gridx = 1;
conTextFieldName.gridy = 0;
conTextFieldName.insets = new Insets(10,0,10,20);
                                        //设置组件四周的间距（上、左、下、右）
topPane.add(textFieldName, conTextFieldName);
```

在上面的代码中，Insets 对象是容器边界的表示形式。它指定容器必须在其各个边缘留出的空间。这个空间可以是边界、空白空间或标题。类似地，对于“密码”标签和密码文本框，布局代码如下所示。

```
//标签"密码"
GridBagConstraints conLabelPassword = new GridBagConstraints();
conLabelPassword.fill = GridBagConstraints.NONE;
labelPassword = new JLabel("密码:");
conLabelPassword.gridx = 0;
conLabelPassword.gridy = 1;
conLabelPassword.anchor = GridBagConstraints.LINE_END;
topPane.add(labelPassword, conLabelPassword);
//密码框
GridBagConstraints conTextFieldPwd = new GridBagConstraints();
conTextFieldPwd.fill = GridBagConstraints.HORIZONTAL;
textFieldPwd = new JPasswordField();
conTextFieldPwd.weighty = 0.5;
conTextFieldPwd.gridx = 1;
conTextFieldPwd.gridy = 1;
conTextFieldPwd.insets = new Insets(0,0,0,20);
topPane.add(textFieldPwd, conTextFieldPwd);
```

至此，topPane 面板布局已经基本完成。对于 bottomPane 面板，布局比较简单，只有两个按钮组件，布局代码如下所示。

```
buttonOk = new JButton("确定");      //创建“确定”按钮
buttonCancel = new JButton("取消"); //创建“取消”按钮
bottomPane.add(buttonOk);           //将“确定”按钮添加到 bottomPane 面板
bottomPane.add(buttonCancel);       //将“取消”按钮添加到 bottomPane 面板
```

两个面板 topPane 和 bottomPane 布局完成以后，需要将它们添加到内容面板中。内容面板默认的布局管理器为 BorderLayout，将 topPane 添加到内容面板的中间区域，将 bottomPane 添加到内容面板的底部，代码如下所示。

```
pane.add(topPane,BorderLayout.CENTER);
pane.add(bottomPane,BorderLayout.PAGE_END);
```

完整的程序代码如示例 16-5 所示。

```
/* 示例 16-5  */
package chapter16;
import java.awt.*;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.UIManager;
public class LoginFrame{
    //添加组件的方法
    public void addComponentsToPane(Container pane) {
        JButton buttonOk,buttonCancel;              //创建按钮对象
        JLabel labelName,labelPassword;             //创建标签对象
        JTextField textFieldName;                   //创建文本框对象
        JPasswordField textFieldPwd;                //创建密码框对象
        JPanel topPane = new JPanel();              //放置标签和文本框的面板
        topPane.setLayout(new GridBagLayout());
                                          //设为网格包布局(GridBagLayout)
        JPanel bottomPane = new JPanel();  //放置"确定"和"取消"按钮的面板
        FlowLayout flowLayout = new FlowLayout();  //创建流动布局管理器对象
        flowLayout.setHgap(20);             //设置流动布局中组件间水平间距
        flowLayout.setVgap(10);             //设置流动布局中组件间垂直间距
        bottomPane.setLayout(flowLayout);
        //标签"账户"
        GridBagConstraints conLabelName = new GridBagConstraints();
        conLabelName.fill = GridBagConstraints.NONE;
        labelName = new JLabel("账户:");
        conLabelName.weightx = 0.2;
        conLabelName.gridx = 0;
        conLabelName.gridy = 0;
        conLabelName.anchor = GridBagConstraints.LINE_END;
        topPane.add(labelName, conLabelName);
        //账户文本框
        GridBagConstraints conTextFieldName = new GridBagConstraints();
        conTextFieldName.fill = GridBagConstraints.HORIZONTAL;
```



```
        textFieldName = new JTextField();
        conTextFieldName.weightx = 0.8;
        conTextFieldName.weighty = 0.5;
        conTextFieldName.gridx = 1;
        conTextFieldName.gridy = 0;
        conTextFieldName.insets = new Insets(10,0,10,20);
        topPane.add(textFieldName, conTextFieldName);
        //标签"密码"
        GridBagConstraints conLabelPassword = new GridBagConstraints();
        conLabelPassword.fill = GridBagConstraints.NONE;
        labelPassword = new JLabel("密码:");
        conLabelPassword.gridx = 0;
        conLabelPassword.gridy = 1;
        conLabelPassword.anchor = GridBagConstraints.LINE_END;
        topPane.add(labelPassword, conLabelPassword);
        //密码框
        GridBagConstraints conTextFieldPwd = new GridBagConstraints();
        conTextFieldPwd.fill = GridBagConstraints.HORIZONTAL;
        textFieldPwd = new JPasswordField();
        conTextFieldPwd.weighty = 0.5;
        conTextFieldPwd.gridx = 1;
        conTextFieldPwd.gridy = 1;
        conTextFieldPwd.insets = new Insets(0,0,0,20);
        topPane.add(textFieldPwd, conTextFieldPwd);
        buttonOk = new JButton("确定");
        buttonCancel = new JButton("取消");
        bottomPane.add(buttonOk);
        bottomPane.add(buttonCancel);
        pane.add(topPane,BorderLayout.CENTER);
        pane.add(bottomPane,BorderLayout.PAGE_END);
    }
    // 创建 GUI 界面并显示
    private void createAndShowGUI() {
        //创建并设置窗口
        JFrame frame = new JFrame("LoginFrame");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //设置内容面板
        addComponentsToPane(frame.getContentPane());
        //显示窗口
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                UIManager.put("swing.boldMetal", Boolean.FALSE);
                                                        //关闭粗体字显示
                new LoginFrame().createAndShowGUI();
            }
        });
    }
}
```

在这个程序中，采用的是 GridBagLayout（网格包布局）布局方式。本示例相对比较

简单，因此进行界面设计还比较容易。如果要设计较为复杂的界面，一般情况下，还是建议使用成熟的集成开发工具。那样会提高开发速度和设计质量。

16.5 小结

本章主要介绍了 Swing 中的程序界面布局方式，包括 BorderLayout、BoxLayout、CardLayout、FlowLayout、GridBagLayout、GridLayout、GroupLayout 和 SpringLayout。

在本章的学习中，读者应将重点放在理解 Swing 程序的布局原理及布局管理器的概念，其中 GroupLayout 和 SpringLayout 两种布局方式读者可能会感到困惑。这两种布局主要是提供给开发工具使用的，因此读者仅做了解即可。在各种布局管理器中，最灵活同时又最复杂的是 GridBagLayout，这也是读者掌握 Java 界面布局管理的一个难点。下一章介绍 Java 程序中的事件处理机制，实现程序和用户的交互。

第17章 事件处理

图形用户界面的程序，需要具备与用户交互的功能。所谓“与用户交互”，也就是说，程序既能显示信息给用户，也能响应用户的操作。例如，在上一章的用户登录验证程序界面中，当用户输入用户名和密码以后，单击【确定】按钮，那么程序就要对这个单击操作做出响应，对用户名和密码进行验证。这个过程，称为“事件处理”。

17.1 事件处理原理

在图形用户界面的程序中，程序需要对用户的操作，如单击鼠标、移动鼠标、键盘输入等，做出响应。用户的这些动作，对于程序来说，会引发相应的事件。那么，什么是事件？事件是如何产生的？程序如何对事件进行响应呢？

所谓事件，是指对鼠标、键盘或其他输入设备的各种操作。如果用户在用户界面上执行了一个操作（如单击某个按钮或敲击键盘），将导致一个事件的发生。

当事件发生后，图形用户界面的程序需要对这种操作进行响应，即对发生的事件进行处理，称为事件处理。如在文本框中输入字符串，在下拉列表中进行选择，单击界面上某个按钮。

在应用事件处理机制的应用程序中，当某个事件产生后，程序会调用事先写好的程序代码进行事件响应。例如，在上一章的用户登录验证程序界面中，要使这个程序能对用户的操作进行响应，那么需要在程序中事先编写好对用户名和密码进行验证的程序代码。当用户输入用户名和密码以后，单击【确定】按钮，就会产生一个单击事件，程序需要调用事先写好的验证代码对这个事件进行处理，即进行密码验证。

17.2 Java 事件处理模型

Java 中的事件处理模型称为授权事件模型，由三个基本要素组成：事件源、事件对象及事件监听器：

- 事件源指的是产生事件的组件或容器，即事件的产生者。当用户在 Java 图形用户界面程序上执行某一个动作（如单击“确定”按钮）时，就导致一个单击事件，“确

定”按钮就是一个事件源。

- 事件指的是在事件源上所执行的动作。在 Java 中，通过对象的形式对事件进行封装与处理。在 Java 中不同的操作对应着不同的事件。当某个事件发生了，Java 会将事件封装到一个事件对象中，这个事件对象记录事件源及处理该事件所需要的各种信息。
- 事件监听器是 Java 中的实现了特定接口的类，专门用来对事件进行处理。

事件源、事件和事件监听器三者之间的关系如图 17.1 所示。

在 Java 程序中实现事件模型，需要执行以下步骤：

① 创建相关事件的事件监听器。事件监听器所对应的接口一般位于包 java.awt.event 和 javax.swing.event 中，而且类名通常以 Listener 结尾。

② 注册监听器。在需要被监听的事件源中登记注册事件监听器，这样在事件源和事件监听器中就建立了监听与被监听的联系。

事件源可能不止产生一种事件，如对于一个按钮，可能产生三种事件：按钮被按下、按钮被释放、按钮被单击。所以在一个事件源上可以注册多个监听器，分别监听不同的事件，如图 17.2 所示。

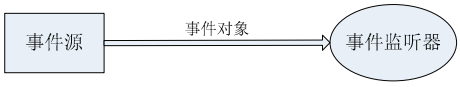


图 17.1　事件处理模型

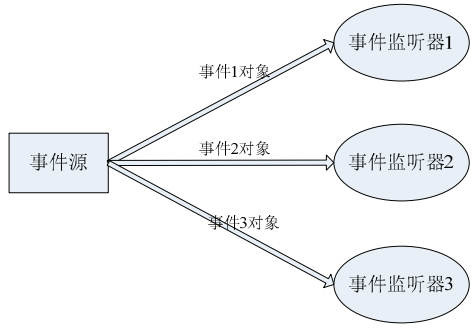


图 17.2　处理多个事件模型

当有事件发生时，Java 虚拟机会产生一个事件对象，在事件对象中记录有处理该事件所需要的各种信息。当事件源接收到事件对象时，就会依次启动在该事件源中注册的事件监听器，并将事件对象传递给相应的事件监听器。这些事件监听器接收到该事件对象，获取事件对象中封装的信息，并对事件进行处理。

在 Java 的事件处理模型中，事件监听器只是简单地等待，直到它收到一个事件。它有两个要求：必须在事件源上进行注册，必须实现接收和处理事件通知的方法。

Java 这种授权事件模型的优点，是处理事件的应用程序和用来产生事件的用户接口程序相分离，一个用户接口程序可以授权一段特定的代码处理一个事件。

17.3　Java 事件与监听器的类型

在图形用户界面的程序中，用户的不同操作，如单击鼠标、移动鼠标、键盘输入等，会引发不同的事件类型，而不同的事件，需要相对应的事件监听器来监听并处理事件。在本节中就来讲解 Java 中有什么类型的事件，不同类型时间需要哪些特定类型事件监听器来处理。



17.3.1　Java 事件类型

在 Java 中，对应不同的事件，有不同类型的事件对象，而所有的事件对象都是从 java.util.EventObject 派生而来的。图 17.3 给出了 Java 中事件类的继承关系。

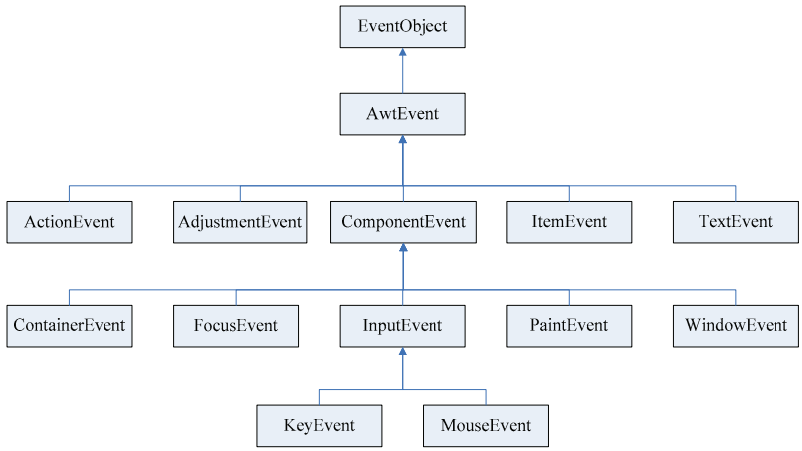


图 17.3　Java 事件类的继承关系图

Java 中的事件类，都位于 java.awt.event 包和 javax.swing.event 包中。从图 17.3 中可以看出，EventObject 是所有事件类的父类。

在 Java 中，事件分为两类：语义事件和低级事件。语义事件指的是用于表达用户动作的事件。例如，“单击按钮”、“移动滚动条”。低级事件是语义事件的基础，如对于“单击按钮”事件，它的低级事件包括一次鼠标键的按下、一系列鼠标的移动及一次鼠标键的释放。所以，鼠标按钮被按下、释放、拖动都是低级事件。在图 17.3 所示的事件类中，属于语义事件类的，如表 17.1 所示。

表 17.1　Java 语义事件类

事　　件	事件对象	含　　义
动作事件	java.awt.event.ActionEvent	对应按钮单击、菜单选择、在文本框中按回车键等
调整事件	java.awt.event.AdjustmentEvent	用户调整滚动条的滑块位置
选项事件	java.awt.event.ItemEvent	复选框的选中状态发生变化
文本事件	java.awt.event.TextEvent	文本域或者文本框中的内容发生改变
列表选择事件	javax.swing.event.ListSelectionEvent	列表框选项发生变化

属于低级事件类的如表 17.2 所示。

表 17.2　Java 低级事件类

事　　件	事件对象	含　　义
组件事件	java.awt.event.ComponentEvent	移动组件、改变组件的大小、显示或隐藏组件。它是所有低级事件类的基类
键盘事件	java.awt.event.KeyEvent	键盘上的一个键被按下或者释放，如通过键盘输入字符
鼠标事件	java.awt.event.MouseEvent	按下、释放鼠标，移动或拖动鼠标
焦点事件	java.awt.event.FocusEvent	组件获得或失去焦点
窗口事件	java.awt.event.WindowEvent	窗口被激活、屏蔽、最小化、最大化或关闭

17.3.2 Java 监听器类型

不同的事件源产生不同的事件，对不同的事件的处理对应于不同的监听器接口，不同的监听器接口有不同的方法。事件监听器对应的接口一般也位于包 java.awt.event 和 javax.swing.event 中。

在包 java.awt.event 和 javax.swing.event 中定义的事件监听器接口的命名一般以 Listener 结尾。这些接口规定了处理相应事件必须实现的基本方法。因此，实际处理事件的事件监听器所对应的类一般是实现这些事件监听器接口的类。Java 中常见的事件监听器类型如表 17.3 所示。

表 17.3 Java 常见事件监听器类型

事件监听器名称	事件监听器接口	事件处理方法	对应的事件
动作事件监听器	java.awt.event.ActionListener	actionPerformed	ActionEvent
调整事件监听器	java.awt.event.AdjustmentListener	adjustmentValueChanged	AdjustmentEvent
选项事件监听器	java.awt.event.ItemListener	itemStateChanged	ItemEvent
文本事件监听器	java.awt.event.TextListener	textValueChanged	TextEvent
列表选择事件监听器	javax.swing.event.ListSelectionListener	valueChanged	ListSelectionEvent
组件事件监听器	java.awt.event.ComponentListener	componentMoved componentHidden componentResized componentShown	ComponentEvent
键盘事件监听器	java.awt.event.KeyListener	keyPressed keyReleased keyTyped	KeyEvent
鼠标事件监听器	java.awt.event.MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent
鼠标移动事件监听器	java.awt.event.MouseMotionListener	mouseDragged mouseMoved	MouseEvent
焦点事件监听器	java.awt.event.FocusListener	focusGained focusLost	FocusEvent
窗口事件监听器	java.awt.event.WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	windowEvent

17.3.3 Java 适配器类

当定义一个监听器类时，必须实现相应的监听器接口及接口中定义的所有方法。从表 17-3 中可以看出，有些接口要求实现相当多的方法，如鼠标事件监听器 MouseListener、窗口事件监听器 WindowListener 等。但在实际应用时，在大多数情况下，只需要处理某一类



型的事件，并不需要对所有的成员方法进行处理。

例如，在某一应用程序中，处理鼠标单击事件时只需要实现 MouseListener.mouseClicked()这一个方法，在这种情况下，去实现对程序员来说毫无意义的其他方法是一件相当烦人的事情。为了让程序员不用实现在接口中定义的所有方法，Java 提供了许多适配器类，这些适配器类已经实现了相应的接口及接口中的全部方法，只是方法全部为空操作。

因此，当程序员在构建自己的监听器类时，可以直接从事件适配器类派生出子类，然后只需重新实现所需要处理的成员方法即可，可省去很多不必要的麻烦。Java 中提供的事件适配器类如表 17.4 所示。

表 17.4　Java 事件适配器类

适配器类	实现事件监听器名称	实现事件监听器接口
抽象类 ComponentAdapter	组件事件监听器	java.awt.event.ComponentListener
抽象类 KeyAdapter	键盘事件监听器	java.awt.event.KeyListener
抽象类 MouseAdapter	鼠标事件监听器	java.awt.event.MouseListener
抽象类 MouseMotionAdapter	鼠标移动事件监听器	java.awt.event.MouseMotionListener
抽象类 FocusAdapter	焦点事件监听器	java.awt.event.FocusListener
抽象类 WindowAdapter	窗口事件监听器	java.awt.event.WindowListener

从表 17.4 可以看出，Java 中提供的事件适配器类均为抽象类。



17.4　处理动作事件

在了解了 Java 的事件处理模型及常见的事件与监听器类型之后，下面学习如何在程序中具体地处理常见的事件，以实现软件与用户的交互。常用的事件有动作事件、焦点事件、鼠标事件和键盘事件。

17.4.1　编写动作事件处理程序的步骤

要使程序能够处理用户界面所触发的动作事件，必须在程序中完成三个步骤：

① 创建一个动作事件监听器类。动作事件监听器类为实现了 ActionListener 接口的 Java 普通类。

② 实现动作事件监听器接口中的一个或一组事件处理方法。ActionListener 接口中，只有一个事件处理方法 actionPerformed()。所以实现了 ActionListener 接口的事件监听器类，只需实现事件处理方法 actionPerformed()即可。

③ 为产生动作事件的组件注册这个事件监听器。通过调用组件的事件监听器添加方法：addActionListener（ActionListener），向一个事件源组件注册动作事件监听器。

17.4.2　处理动作事件的过程

动作事件指的是按钮单击、菜单选择，或在文本框中按回车键等动作所触发的事件。动作事件由 ActionEvent 类捕获并封装事件的详细信息。当动作事件被触发时，封装的事件信息（ActionEvent 对象）会传递给已经注册的动作事件监听器，在监听器中就可以使用

ActionEvent 对象的方法来获得事件源的相关信息。ActionEvent 类有如下两个比较常用的方法。

- getSource()：返回触发此次事件的事件源组件对象，返回值类型为 Object；
- getActionCommand()：返回与当前动作相关的命令字符串，返回值类型为 String。

动作事件的监听器类必须实现 ActionListener 接口。在 ActionListener 接口中只有一个抽象方法 actionPerformed()，用来处理事件。所以，实现了 ActionListener 接口的监听器类，必须实现 actionPerformed()方法。ActionListener 接口的定义如下。

```
public interface ActionListener extends EventListener{
    public void actionPerformed(ActionEvent e);
}
```

为产生事件的组件注册动作事件监听器，需要用到组件的 addActionListener (ActionListener listener)方法。

17.4.3 处理按钮触发的动作事件

单击按钮时，会触发相应的动作事件。下面通过一个实例程序 ButtonActionEventDemo.java，演示动作事件的触发及事件处理过程。

【例 17.1】编写一个用来演示由按钮触发动作事件的示例。程序的功能如图 17.4～图 17.7 所示。



图 17.4 登录界面



图 17.5 单击“登录”按钮后

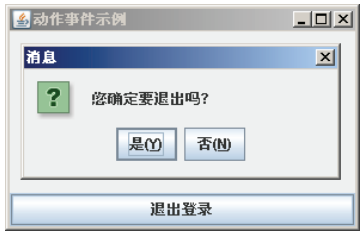


图 17.6 单击“登录”按钮后



图 17.7 单击“退出登录”按钮后

程序功能要求如下。

- 程序启动后，用户单击【登录】按钮，弹出成功登录信息提示框，同时按钮上的文本由“登录”变为“退出登录”，如图 17.4、图 17.5 所示。
- 当用户单击【退出登录】按钮后，弹出退出登录信息提示框，如图 17.6 所示。
- 当用户选择【是】确定退出后，按钮上的文字由“退出登录”变为“登录”，如 17.7 所示。

实现此程序的步骤如下。



① 在 D 盘根目录下，创建文件夹 chapter17，并在 chapter17 目录下，创建 java 源文件 ButtonActionEventDemo.java；

② 使用记事本，打开 ButtonActionEventDemo.java 源文件进行编辑，输入以下代码：

```
package chapter17;      //创建 chapter17 包
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



程序放在包 chapter17 中；需要导入 java.awt 包、java.awt.event 包和 javax.swing 包。

③ 接着编写“登录”按钮的事件监听器类 ButtonActionListener，这个类实现了动作事件监听器接口 ActionListener。代码如下。

```
//实现监听器类 ButtonActionListener
class ButtonActionListener implements ActionListener{
   public void actionPerformed(ActionEvent e){
        JButton button=(JButton)e.getSource();      //获得触发此次事件的事件源组件对象
        String buttonName=e.getActionCommand();     //获得触发此次事件的按钮的命令字符串
        //显示单击信息
        if(buttonName.equals("登录")){  //如果按钮上的文本为“登录”
            button.setText("退出登录");  //首先将按钮上的文本更改为“退出登录”
            JOptionPane.showMessageDialog(null, "您已经成功登录!", "消息",
                                JOptionPane.INFORMATION_MESSAGE);
        }else{
            int answer=JOptionPane.showConfirmDialog(null, "您确定要退出吗?", "消息",
                                                    JOptionPane.YES_NO_OPTION);
            //如果用户单击"确定"退出,更改按钮上的文字
            if(answer==0)
                button.setText("登录");
        }
   }
}
```

在上面的代码中，注意按钮上文字的改变。当单击“登录”按钮以后，按钮上的文字变为“退出登录”；当用户单击【确定】按钮退出时，按钮上的文字变为“登录”。

④ 编写程序窗体界面类 ActionEventFrame，该类继承自 JFrame 类。在该类中添加了一个按钮 OKButton，并为此按钮添加一个动作事件监听器。具体代码如下。

```
//编写应用程序 ActionEventFrame
class ActionEventFrame extends JFrame{
    //构造函数
   ActionEventFrame(){
        super();                                        //首先执行超类的构造器
        setTitle("动作事件示例");                        //设置标题
        setBounds(100,100,300,200);                     //设置窗体大小
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);    //设置关闭窗体时退出程序
```

```
        JButton OKButton=new JButton();                    //创建一个按钮
        OKButton.setText("登录");                          //设置按钮表面文本
        OKButton.addActionListener(new ButtonActionListener());
                                                  //为"登录"按钮添加动作监听器
        getContentPane().add(OKButton,BorderLayout.SOUTH);
                                                  //向面板中添加按钮组件
        this.setVisible(true);                    //显示窗体
    }
}
```

在构造器中主要完成对象的初始化及为按钮添加动作监听器。

⑤ 创建主程序 ActionEventDemo，具体代码如下。

```
//编写主程序
public class ButtonActionEventDemo{
    public static void main(String[] args){
        ActionEventFrame actionEventFrame=new ActionEventFrame(); //启动窗体
    }
}
```

⑥ 使用下面的命令编译并运行主程序 ButtonActionEventDemo.java：

```
javac -d . ButtonActionEventDemo.java
java chapter17.ButtonActionEventDemo
```

这时可以看到如图 17.4 所示的程序运行界面。单击屏幕下边的“登录”按钮，弹出登录成功的消息对话框，如图 17.5 所示。分别如图 17.6 和图 17.7 所示进行操作，观察程序的响应过程。

17.4.4 处理选择菜单时触发的事件

单击菜单中的菜单项时，会触发相应的动作事件。下面通过一个 MenuActionEventDemo.java 实例程序，演示选择菜单项时动作事件的触发及事件处理过程。

【例 17.2】编写一个用来演示由选择菜单时触发动作事件的示例。程序的功能如图 17.8 和图 17.9 所示。

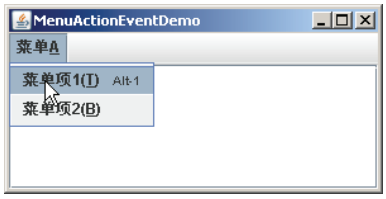


图 17.8 登录界面图



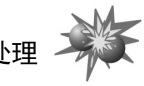
图 17.9 在文本框中显示事件信息

程序功能要求如下。

- 程序启动后，用户选择【菜单 A】|【菜单项 1（T）】命令，或者同时按下 Alt+1 键，如图 17.8 所示。
- 程序会检测到用户单击菜单项的事件，并解析事件信息，在文本框中显示，如图 17.9 所示。

实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter17，并在 chapter17 目录下，创建 java 源文



件 MenuActionEventDemo.java；

② 使用记事本，打开 MenuActionEventDemo.java 源文件进行编辑，输入以下代码：

```
package chapter17;      //创建 chapter17 包
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



程序放在包 chapter17 中；需要导入 java.awt 包、java.awt.event 包和 javax.swing 包。

③ 接着编写菜单项的事件监听器类 MenuActionEventDemo，这个类实现了动作事件监听器接口 ActionListener，因此这个类本身又是一个动作事件监听器类。代码如下。

```
//实现监听器类 ButtonActionListener
public class MenuActionEventDemo implements ActionListener{
    JTextArea output;
    JScrollPane scrollPane;
    String newline = "\n";
    //创建菜单栏的方法
    public JMenuBar createMenuBar() {
        JMenuBar menuBar;                    //声明一个菜单栏
        ……                                   //这里为创建菜单栏的代码，下一步完成
        return menuBar;                      //返回创建的菜单栏
    }
     //将组件添加到面板 pane 中的访求
     public void addComponentsToPane(Container pane) {
         output = new JTextArea(5, 25);
        output.setEditable(false);
        scrollPane = new JScrollPane(output);
        pane.add(scrollPane, BorderLayout.CENTER);    //将文本框添加到内容面板中
     }
     public void actionPerformed(ActionEvent e) {
         ……                                          //这里为事件处理代码
     }
     /*  创建 GUI 界面并显示 */
     public void createAndShowGUI() {
        ……                          //这里为创建并显示 GUI 界面的代码，后面继续完成
     }
     public static void main(String[] args) {
        ……                          //这里启动线程，调用创建并显示 GUI 界面的方法
     }
}
```

其中 MenuActionEventDemo 类既是主程序类，也是实现了动作事件监听器接口的监听器类。

④ 自定义一个创建菜单栏的方法 createMenuBar()，这个方法返回一个创建好的菜单栏，并给菜单栏中的菜单项注册动作事件监听器。代码如下。

```
public JMenuBar createMenuBar() {
    JMenuBar menuBar;                        //声明一个菜单栏
    JMenu menu;                              //声明一个菜单
    JMenuItem menuItem;                      //声明一个菜单项
```

```
        menuBar = new JMenuBar();              //创建菜单栏
        //构建菜单 A.
        menu = new JMenu("菜单 A");
        menu.setMnemonic(KeyEvent.VK_A);       //设置快捷键
        menuBar.add(menu);                     //将菜单添加到菜单栏中
        //构造菜单 A 中的菜单项 1 和菜单项 2
        menuItem = new JMenuItem("菜单项 1(T)",KeyEvent.VK_T);
        //为菜单项 1 设置加速键：Alt+1
        menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_1,
ActionEvent.ALT_MASK));
        menuItem.addActionListener(this);      //将当前对象作为注册为自身的监听器
        menu.add(menuItem);                    //几菜单中添加菜单项
        menuItem = new JMenuItem("菜单项 2(B)",KeyEvent.VK_B);
        menuItem.addActionListener(this);      //将当前对象作为注册为自身的监听器
        menu.add(menuItem);
        return menuBar;                        //返回创建的菜单栏
    }
```

将创建菜单栏的任务放在方法 createMenuBar()中创建。当调用 createMenuBar()方法时，会返回创建好的一个菜单栏，这使得程序代码更加清晰，封装性更好。

⑤ 自定义一个方法 createAndShowGUI，用来创建窗体并显示 GUI 界面，代码如下所示。

```
    /* 创建 GUI 界面并显示 */
    public void createAndShowGUI() {
        //创建并设置窗口
        JFrame frame = new JFrame("MenuActionEventDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //设置菜单栏和内容面板
        frame.setJMenuBar(this.createMenuBar());
        addComponentsToPane(frame.getContentPane());
        //显示窗口
        frame.setSize(300, 150);
        frame.setVisible(true);
    }
```

在 createAndShowGUI()方法中创建 GUI 界面，并设置菜单栏。

⑥ 添加一个事件处理方法 actionPerformed()，在该方法中检测事件信息，代码如下。

```
    public void actionPerformed(ActionEvent e) {
        JMenuItem source = (JMenuItem)(e.getSource());      //获得事件源对象
        String s = "检测到动作事件."
                 + newline
                 + "    事件源: " + source.getText()
                 + ".它是" + getClassName(source) + "的一个实例.";
        output.append(s + newline);                    //在文本域中添加事件消息
        output.setCaretPosition(output.getDocument().getLength());
                                                       //将光标放在插入内容的最后
    }
    // 返回不带包名信息的类的名字.
    protected String getClassName(Object o) {
        String classString = o.getClass().getName();    //获得对象 o 所属的类名
        int dotIndex = classString.lastIndexOf(".");
```

```
    return classString.substring(dotIndex+1);          //返回类名字
}
```

当监听器监测到事件时，会通过调用事件对象的 getSource()方法解析事件，获得事件源，然后将事件信息追加到文本域中。每次追加信息以后，通过调用文本域对象的 setCarePosition()方法，将光标放在追加的内容的最后。这样当内容超过显示区域时，会自动滚动，总是将最新插入的信息显示出来。

⑦ 编写 main()方法，具体代码如下。

```
//编写主程序
public static void main(String[] args) {
    new MenuActionEventDemo().createAndShowGUI();
}
```

⑧ 使用以下命令编译并运行主程序 MenuActionEventDemo.java：

```
javac -d . MenuActionEventDemo.java
java chapter17. MenuActionEventDemo
```

这时，会得到如图 17.8 所示的程序界面。选择单击不同的菜单，这时在文本域中会看到相应的事件响应信息，如图 17.9 所示。

17.4.5　处理文本框触发的事件

在文本框中输入内容后，单击回车键时，会触发相应的动作事件。下面通过一个实例程序 TextFieldActionEventDemo.java，演示使用文本框时动作事件的触发及事件处理过程。

【例 17.3】编写一个用来演示文本框触发动作事件的示例。程序的功能如图 17.10 和图 17.11 所示。



图 17.10　在第一个文本框中输入内容



图 17.11　在第二文本框中响应事件

程序功能要求如下。

- 程序启动后，在第一个文本框中输入“大家好”，然后按下 Enter 键，如图 17.10 所示。
- 程序会检测到用户在文本框中的事件，在第二个文本框中将第一个文本框中输入的内容显示出来，如图 17.11 所示。

实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter17，并在 chapter17 目录下，创建 java 源文件 TextFieldActionEventDemo.java；

② 使用记事本，打开 TextFieldActionEventDemo.java 源文件进行编辑，输入以下代码：

```
package chapter17;      //创建 chapter17 包
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



程序放在包 chapter17 中；需要导入 java.awt 包、java.awt.event 包和

javax.swing 包。

③ 接着编写一个自定义文本框 TargetTextField，这个类实现了动作事件监听器接口 ActionListener，所以它既是一个文本框，也是一个动作事件监听器。代码如下。

```
//自定义文本框,并将其实现为监听器
class TargetTextField extends JTextField implements ActionListener{
        TargetTextField(){                                    //不带参数的构造器
            super();
        }
        TargetTextField(int length){                          //带有一个 int
型参数的构造器
            super(length);
        }
        public void actionPerformed(ActionEvent e){
            JTextField txtField =(JTextField)e.getSource();        //获得
触发此次事件的事件源组件对象
            this.setText(txtField.getText());                  //将源文本
框中的文本写入当前文本框
        }
}
```

自定义的文本框继承自 Swing 的 JTextField 组件，并将其实现为动作事件监听器。

④ 编写程序窗体界面类 TextFieldActionEventFrame，该类继承自 JFrame 类。在该类中添加了两个文本框 sourceTextField 和 targetTextField，其中 targetTextField 为上一步定义的 TargetTextField 的对象，所以将它注册为 sourceTextField 的动作事件监听器。具体代码如下所示。

```
//编写程序窗体界面类 TextFieldActionEventFrame
class TextFieldActionEventFrame extends JFrame{
    TextFieldActionEventFrame(){
        super();
        setTitle("文本框动作事件示例");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JTextField sourceTextField = new JTextField(10);   //源文本框
        TargetTextField targetTextField = new TargetTextField(10);
                                                           //目标文本框
        sourceTextField.addActionListener(targetTextField);
                                                           //注册动作监听器
        JLabel label = new JLabel("=>");                   //创建一个标签对象
        JPanel panel = new JPanel();                       //创建容器面板
        panel.add(sourceTextField);                        //向面板中添加源文本框
        panel.add(label);                                  //向面板中添加标签
        panel.add(targetTextField);                        //向面板中添加目标文本框
        this.getContentPane().add(panel,BorderLayout.CENTER);
                                                          //将面板组件添加到内容窗格中
        this.pack();
        this.setVisible(true);                            //显示窗体
    }
}
```

在程序界面类中，实现组件的初始化和布局。



⑤ 创建主程序 TextFieldActionEventDemo，具体代码如下。

```
//编写主程序
public class TextFieldActionEventDemo {
    public static void main(String[] args){
        TextFieldActionEventFrame textFieldActionEventFrame =
                            new TextFieldActionEventFrame();    //启动窗体
    }
}
```

⑥ 使用下面的命令编译并运行主程序 TextFieldActionEventDemo.java。

```
javac -d . TextFieldActionEventDemo.java
java chapter17. TextFieldActionEventDemo
```

这时，会得到如图 17.10 所示的程序界面。在左边的文本框中输入内容，并按回车键，在右边的文本框中会出现左边文本框中的内容，如图 17.11 所示。



17.5　处理选项事件

选项事件指的是当多个单选或复选按钮的状态发生变化时触发的事件。选项事件通常发生在用户改变复选框、单选按钮或组合框的选择项时。本节介绍选择事件处理的步骤、过程和事件处理。

17.5.1　编写选项事件处理程序的步骤

要使程序能够处理用户界面所触发的选项事件，必须在程序中完成三个步骤：

① 创建一个选项事件监听器类。选项事件监听器类为实现了 java.awt.event.ItemListener 接口的 Java 普通类。

② 实现选项事件监听器接口中的一个或一组事件处理方法。ItemListener 接口中，只有一个事件处理方法 itemStateChanged()。所以实现了 ItemListener 接口的事件监听器类，只需实现事件处理方法 itemStateChanged()即可。

③ 为产生选项事件的组件注册这个事件监听器。通过调用组件的事件监听器添加方法：addItemListener（ItemListener），向一个事件源组件注册选项事件监听器。

17.5.2　处理选项事件的过程

选项事件指的是当用户改变复选框、按钮等选择项的状态时所触发的事件。选项事件由 java.awt.event.ItemEvent 类捕获并封装事件的详细信息。当选项事件被触发时，封装的事件信息（ItemEvent 对象）会传递给已经注册的选项事件监听器，在监听器中就可以使用 ItemEvent 对象的方法来获得事件源的相关信息。ItemEvent 类有如下两个比较常用的方法。

- getItemSelectable()：返回触发此次事件的事件源组件对象，返回值类型为 Object；
- getStateChange()：返回值为 ItemEvent.DESELECTED 常量或 ItemEvent.SELECTED 常量。

选项事件的监听器类必须实现 ItemListener 接口。在 ItemListener 接口中只有一个抽象方法 itemStateChanged，用来处理事件。所以，实现了 ItemListener 接口的监听器类，必须

实现 itemStateChanged 方法。ItemListener 接口的定义如下所示。

```
public interface ItemListener extends EventListener{
    public void itemStateChanged (ItemEvente);
}
```

为产生事件的组件注册选项事件监听器，需要用到组件的 addItemListener (ItemListener listener)方法。

17.5.3 处理选择项状态改变所触发的事件

单击复选框或单选按钮改变其状态时，会触发相应的选项事件。下面通过一个实例程序 ButtonActionEventDemo.java，演示使用单选按钮时处理选择项状态改变事件的触发及事件处理过程。

【例 17.4】编写一个用来演示由选择不同单选按钮触发选项事件的示例。程序的功能如图 17.12 所示。



图 17.12 响应选项事件

程序功能要求：程序启动后，用户分别单击“鹦鹉”、“狗狗”和“小花猫”单选按钮，下面的标签分别显示相应的图片，如图 17.12 所示。实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter17，并在 chapter17 目录下，创建 java 源文件 ItemActionEventDemo.java；

② 使用记事本，打开 ItemActionEventDemo.java 源文件进行编辑，输入以下代码：

```
package chapter17;      //创建 chapter17 包
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



程序放在包 chapter17 中；需要导入 java.awt 包、java.awt.event 包和 javax.swing 包。

③ 接着编写自定义标签 PictureLabel，并让其作为单选按钮的选项监听器类，这个类实现了选项事件监听器接口 ItemListener。代码如下。

```
//自定义标签 PictureLabel,并实现监听器类 ItemListener
class PictureLabel extends JLabel implements ItemListener{
    public PictureLabel(){
        this.setIcon(createImageIcon("images/Bird.gif")); //设置标签默认图像
        this.setBorder(BorderFactory.createEmptyBorder(10,0,10,0));
                                                          //设置标签的边框
    }
```

```
    //选项事件处理方法
    public void itemStateChanged(ItemEvent e){
        ImageIcon icon = null;
        JRadioButton sourceButton = (JRadioButton)e.getItemSelectable();
                                                    //获得事件源
        if(sourceButton.getText().equals("鹦鹉"))   //如果用户选择的是“鹦鹉”
            icon = createImageIcon("images/Bird.gif"); //将图标设为鹦鹉图标
        else if(sourceButton.getText().equals("狗狗"))
            icon = createImageIcon("images/Dog.gif");
        else
            icon = createImageIcon("images/Cat.gif");

        this.setIcon(icon);                         //设置标签中显示的图像
        this.validate();                            //使更新过后的标签有效
    }
    /** 返回一个 ImageIcon, 如果文件路径无效，返回 null. */
    protected static ImageIcon createImageIcon(String path) {
        java.net.URL   imgURL   =   PictureLabel.class.getResource(path);
//获得图像文件的路径
        if (imgURL != null) {               //如果是有效的路径
            return new ImageIcon(imgURL);   //创建并返回图像对象
        } else {                            //否则，输出提示信息，并返回 null
            System.err.println("找不到文件: " + path);
            return null;
        }
    }
}
```

上面的 PictureLabel 类继承自 swing 中的 JLabel 类，并实现了 ItemListener 接口，作为单选按钮的事件监听器。所以在 PictureLabel 类中要实现 itemStateChanged()方法，来响应单选按钮事件。另外，将获得图像的操作放在一个单独的方法 createImageIcon()中，方法调用结束后会返回所创建的 ImageIcon 对象。如果不能正确加载图像文件，则返回 null 对象。这使得代码更加清晰。

④ 编写程序窗体界面类 ItemEventFrame，该类继承自 JFrame 类。在该类中添加了三个单选按钮，并为此三个按钮添加同一个选项事件监听器。具体代码如下所示。

```
//编写程序窗体类 ItemEventFrame
class ItemEventFrame extends JFrame{
   ItemEventFrame(){
        super();
        setTitle("选项事件示例");              //设置窗体标题
        setBounds(100,100,300,200);          //设置窗体出现的位置和大小
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                              //设置当关闭窗体时自动退出程序
        //创建三个单选按钮
        JRadioButton rb1 = new JRadioButton("鹦鹉",true);
        JRadioButton rb2 = new JRadioButton("狗狗");
        JRadioButton rb3 = new JRadioButton("小花猫");
        //创建单选按钮组
        ButtonGroup bg = new ButtonGroup();
        bg.add(rb1);
```

```
        bg.add(rb2);
        bg.add(rb3);
        //将单选按钮添加到窗口面板中
        JPanel buttonPane = new JPanel();
        buttonPane.add(rb1);
        buttonPane.add(rb2);
        buttonPane.add(rb3);
        //创建自定义标签对象
        JPanel centerPane = new JPanel();
        PictureLabel pLabel = new PictureLabel();
        centerPane.add(pLabel);
        //为单选按钮添加监听器.都共享同一个监听器
        rb1.addItemListener(pLabel);
        rb2.addItemListener(pLabel);
        rb3.addItemListener(pLabel);
        //添加组件到窗体的内容窗格中
        this.getContentPane().add(buttonPane,BorderLayout.PAGE_START);
        this.getContentPane().add(centerPane,BorderLayout.CENTER);
        //显示窗体
        this.pack();
        this.setVisible(true);
    }
}
```

在程序窗体类 ItemEventFrame 的构造器中，完成组件对象的创建和初始化，并完成布局。在本例中，所有的单选按钮都共享同一个监听器，所以在监听器中要判断监听到的事件信息来自哪一个单选按钮。三个单选按钮需要添加到同一个按钮组中。

⑤ 创建主程序 ItemEventDemo，具体代码如下。

```
//编写主程序
public class ItemEventDemo{
    public static void main(String[] args){
        ItemEventFrame itemEventFrame=new ItemEventFrame();
    }
}
```

⑥ 使用下面的命令编译并运行主程序 ItemEventDemo.java。

```
javac -d . ItemEventDemo.java
java chapter17. ItemEventDemo
```

这时，会得到如图 17.12 所示的程序界面。试着选择不同的单选按钮，观察事件响应过程。

17.6 处理列表选择事件

列表选择事件指的是当在 JList 列表框中选择时触发的事件。本节介绍列表事件处理程序的步骤、事件处理的过程和列表选择事件的处理方法。

17.6.1 编写列表事件处理程序的步骤

要使程序能够处理用户界面所触发的列表选择事件，必须在程序中完成三个步骤：

① 创建一个列表选择事件监听器类。列表选择事件监听器类为实现了 ListSelectionListener 接口的 Java 普通类。

② 实现列表选择事件监听器接口中的一个或一组事件处理方法。ListSelectionListener 接口中，只有一个事件处理方法 valueChanged()。所以实现了 ListSelectionListener 接口的事件监听器类，只需实现事件处理方法 valueChanged()即可。

③ 为产生列表选择事件的列表组件注册这个事件监听器。通过调用列表组件的事件监听器添加方法：addListSelectionListener（ListSelectionListener），向一个事件源组件注册列表选择事件监听器。

17.6.2　处理列表选择事件的过程

列表选择事件指的是当用户在列表中选择一项或多项时所触发的事件。列表选择事件由 javax.swing.event.ListSelectionEvent 类捕获并封装事件的详细信息。当列表选择事件被触发时，封装的事件信息（ListSelectionEvent 对象）会传递给已经注册的选项事件监听器，在监听器中就可以使用 ListSelectionEvent 对象的方法来获得事件源的相关信息。ListSelectionEvent 类有如下三个比较常用的方法。

- getSource()：返回触发此次事件的事件源组件对象，返回值类型为 Object。
- getFirstIndex()：返回多项选择中的第一项的索引值。
- getLastIndex()：返回多项选择中的最后一项的索引值。

选项事件的监听器类必须实现 ListSelectionListener 接口。在 ListSelectionListener 接口中只有一个抽象方法 valueChanged()，用来处理事件。所以，实现了 ListSelectionListener 接口的监听器类，必须实现 valueChanged()方法。ListSelectionListener 接口的定义如下。

```
public interface ListSelectionListener extends EventListener{
    public void valueChanged (ListSelectionEvent e);
}
```

为产生事件的组件注册列表选择事件监听器，需要用到组件的 addListSelectionListener (ListSelectionListener listener)方法。

17.6.3　处理选择列表项时所触发的事件

在列表框中选择列表项时，会触发相应的列表选择事件。下面通过一个实例 17.5，演示列表选择事件的处理过程。

【例 17.5】编写一个用来演示由选择列表框中列表项所触发选项事件的示例。程序的功能如图 17.13 所示。

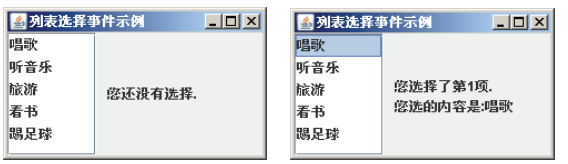


图 17.13　响应列表选择事件

程序功能要求：程序启动后，用户选择左边列表中不同的项，右边的标签就显示所选择的内容，如图 17.13 所示。实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter17，并在 chapter17 目录下，创建 java 源文件 ListEventDemo.java；

② 使用记事本，打开 ListEventDemo.java 源文件进行编辑，输入以下代码：

```
package chapter17;      //创建 chapter17 包
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;     //ListSelectionListener 在此包中定义
```



程序放在包 chapter17 中；需要导入 java.awt 包、java.awt.event 包、javax.swing 和 javax.swing.event.*包。

③ 编写自定义标签 InfoLabel，并让其作为列表的监听器类，这个类实现了列表选择事件监听器接口 ListSelectionListener。代码如下。

```
//自定义标签 PictureLabel,并实现监听器类 ListSelectionListener
class InfoLabel extends JLabel implements ListSelectionListener {
    public InfoLabel(){
        this.setText("您还没有选择.");          //设置自定义标签初始显示的文本
        this.setBorder(BorderFactory.createEmptyBorder(0,10,0,0));
                                                //设置自定义标签的边框
        this.setPreferredSize(new Dimension(150, 50));
                                                //设置自定义标签的首选大小
    }
    //实现 ListSelectionListener 中的事件处理方法
    public void valueChanged (ListSelectionEvent e){
        JList list = (JList)e.getSource(); //获得事件源
        this.setText("");                       //每次响应事件后先清空标签上文本
        String content = "<html>您选择了第" + (list.getSelectedIndex()+1) + "项." +
                         "<br/>您选的内容是:" + list.getSelectedValue();

        this.setText(content);                  //重新设置自定义标签显示的文本
    }
}
```

自定义的类 InfoLabel 继承自 swing 中的 JLabel 类，并实现了 ListSelectionListener 事件监听器接口，作为列表的事件监听器类，所以在此类中要实现事件处理方法 valueChanged()。

④ 编写程序窗体界面类 ListEventFrame，该类继承自 JFrame 类。在该类中添加了一个列表和一个标签（标签同时作为列表的选择事件监听器）。具体代码如下。

```
//编写程序窗体类 ListEventFrame
class ListEventFrame extends JFrame{
   ListEventFrame(){
        super();
        setTitle("列表选择事件示例");
        setBounds(100,100,250,110);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //创建列表
        String[] hoppy = {"唱歌","听音乐","旅游","看书","踢足球"};
        JList list = new JList(hoppy);                          //创建列表对象
```

```
            JScrollPane scrollPane = new JScrollPane(list);    //将列表放在滚
动面板中
            scrollPane.setPreferredSize(new Dimension(80, 110));
            InfoLabel infoLabel = new InfoLabel();             //创建作为监听
器的自定义标签
            //添加事件监听器
            list.addListSelectionListener(infoLabel);
            //添加组件到窗体的内容窗格中
            this.getContentPane().add(scrollPane,BorderLayout.CENTER);
            this.getContentPane().add(infoLabel,BorderLayout.LINE_END);
            //显示窗体
            this.pack();
            this.setVisible(true);
        }
    }
```

在程序窗体类 ListEventFrame 的构造器中实现组件对象的初始化和界面布局。

⑤ 创建主程序 ListEventDemo，具体代码如下。

```
//编写主程序
public class ListEventDemo{
    public static void main(String[] args){
        ListEventFrame listEventFrame = new ListEventFrame();
    }
}
```

⑥ 使用下面的命令编译并运行主程序 ListEventDemo.java。

```
javac -d . ListEventDemo.java
java chapter17. ListEventDemo
```

这时会看到如图 17.13 所示程序界面。试着在左边的列表中选择不同的项，观察右边标签中显示的文本信息。

17.7　处理焦点事件

焦点事件指的是当在组件获得或失去焦点时触发的事件。组件只有获得焦点时，才能响应操作。Swing 中所有的组件都能产生焦点事件。

17.7.1　编写焦点事件处理程序的步骤

要使程序能够处理用户界面所触发的焦点事件，必须在程序中完成三个步骤：

① 创建一个焦点事件监听器类。焦点事件监听器类为实现了 FocusListener 接口的 Java 普通类。

② 实现焦点事件监听器接口中的一个或一组事件处理方法。FocusListener 接口中，有两个事件处理方法：focusGained()和 focusLost()。它们分别在组件获得或失去焦点时被触发。所以实现了 FocusListener 接口的事件监听器类，必须实现这两个事件处理方法。

③ 为产生焦点事件的组件注册这个事件监听器。通过调用组件的事件监听器添加方法：addFocusListener（FocusListener），向一个事件源组件注册焦点事件监听器。

17.7.2 处理焦点事件的过程

焦点事件指的是当在组件获得或失去焦点时触发的事件。焦点事件由 java.awt.event.FocusEvent 类捕获并封装事件的详细信息。当焦点事件被触发时，封装的事件信息（FocusEvent 对象）会传递给已经注册的选项事件监听器，在监听器中就可以使用 FocusEvent 对象的方法来获得事件源的相关信息。FocusEvent 类有一个比较常用的方法。

- getSource()：返回触发此次事件的事件源组件对象，返回值类型为 Object。

焦点事件的监听器类必须实现 FocusListener 接口。在 FocusListener 接口中有两个事件处理方法：focusGained()和 focusLost()，分别用来处理组件获得焦点事件和失去焦点事件。所以，实现了 FocusListener 接口的监听器类，必须实现 focusGained()方法和 focusLost()方法。FocusListener 接口的定义如下。

```
public interface FocusListener extends EventListener{
    public void focusGained (FocusEvent e); //当组件获得焦点时将触发该方法
    publid void focusLost(FocusEvent e);    //当组件失去焦点时将触发该方法
}
```

为产生事件的组件注册列表选择事件监听器，需要用到组件的 addFocusListener (FocusListener listener)方法。

17.7.3 处理组件获得或失去焦点时所触发的事件

在组件如文本框中获得或失去焦点时，会触发相应的焦点事件。所有的 Swing 组件都有获得焦点事件和失去焦点事件。下面的程序示例演示了焦点事件的用法。



图 17.14　响应焦点事件

【例 17.6】编写一个用来演示由文本框触发焦点事件的示例。程序的功能如图 17.14 所示。

程序功能要求：

- 程序启动后，用户在左边文本框中输入账户，如“admin”，然后在右边的密码文本框中单击鼠标，这时密码文本框可以获得焦点，如图 17.14 所示。
- 再次在左边的【账户】文本框中单击鼠标，使其获得信息，这时【账户】文本框应自动清空。
- 在【账户】文本框不输入任何内容的情况下，无法移走焦点。

实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter17，并在 chapter17 目录下，创建 java 源文件 FocusEventDemo.java；

② 使用记事本，打开 FocusEventDemo.java 源文件进行编辑，输入以下代码：

```
package chapter17;      //创建 chapter17 包
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



程序放在包 chapter17 中；需要导入 java.awt 包、java.awt.event 包、



javax.swing 包。

③ 定义主应用程序 FocusEventDemo 类，在该类中定义一个内部类 TextFieldFocus 作为焦点事件监听器。这个类实现了焦点事件监听器接口 FocusListener。代码如下。

```
public class FocusEventDemo extends JFrame{
    JTextField txtName;
    //编写焦点监听器类 TextFieldFocus,它是 FocusEventDemo 类的内部类
    class TextFieldFocus implements FocusListener {
        public void focusGained(FocusEvent e){      //当组件获得焦点时
            txtName.setText("");                    //将 txtName 文本框清空
        }
        public void focusLost(FocusEvent e){        //当组件推动焦点时
            if(txtName.getText().trim().length()==0){  //如果账户为空
                txtName.requestFocusInWindow();     //强制 txtName 文本框重新
获得焦点
            }
        }
    }
}
```

在程序窗体类中，通过定义一个实现 FocusListener 接口的内部类作为焦点事件监听器。

④ 编写构造方法 FocusEventDemo()，为窗体依次添加两个标签和两个文本框，并为账户文本框添加焦点监听器。具体代码如下。

```
//构造方法
   FocusEventDemo(){
        super();
        setTitle("焦点事件示例");
        setBounds(100,100,100,50);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //创建组件
        JLabel label1 = new JLabel("账户:");
        JLabel label2 = new JLabel("密码:");
        txtName = new JTextField(10);
        JTextField txtPwd = new JTextField(10);
        //添加事件监听器
        txtName.addFocusListener(new TextFieldFocus());
        //添加组件到窗体的内容窗格中
        this.getContentPane().setLayout(new FlowLayout());
        this.getContentPane().add(label1);
        this.getContentPane().add(txtName);
        this.getContentPane().add(label2);
        this.getContentPane().add(txtPwd);
        //显示窗体
        this.pack();
        this.setVisible(true);
   }
```

在构造器中完成组件对象的初始化和界面布局。

⑤ 创建主程序 FocusEventDemo，具体代码如下。

```
public static void main(String[] args){
        FocusEventDemo focusEventDemo = new FocusEventDemo();
```

```
}
```

⑥ 使用下面的命令编译并运行主程序 FocusEventDemo.java。

```
javac -d . FocusEventDemo.java
java chapter17. FocusEventDemo
```

这时会看到如图 17.14 所示程序界面。试着在左边的文本框中不输入内容，然后用鼠标单击右边的文本框，试图将焦点移到右边的文本框，这时会发现无法移走焦点。

17.8 处理键盘事件

键盘事件指的是当在键盘上击键时触发的事件，例如在文本框中输入文本时，或者在键盘上击功能键时。在很多应用程序中，如文本编辑器、游戏等，经常需要处理很多键盘事件。

17.8.1 编写键盘事件处理程序的步骤

要使程序能够处理键盘事件，必须在程序中完成三个步骤：

① 创建一个键盘事件监听器类。键盘事件监听器类为实现了 KeyListener 接口的 Java 普通类。

② 实现键盘事件监听器接口中的一个或一组事件处理方法。KeyListener 接口中，有三个事件处理方法：keyTyped()、keyPressed()和 keyReleased()。它们分别用来处理击键事件、键被按下和释放事件。所以实现了 KeyListener 接口的事件监听器类，必须实现这三个事件处理方法。

③ 为产生键盘事件的组件注册这个事件监听器。通过调用组件的事件监听器添加方法：addKeyListener（KeyListener），向一个事件源组件注册键盘事件监听器。

17.8.2 处理键盘事件的过程

键盘事件由 java.awt.event. KeyEvent 类捕获并封装事件的详细信息。当键盘事件被触发时，封装的事件信息（KeyEvent 对象）会传递给已经注册的键盘事件监听器，在监听器中就可以使用 KeyEvent 对象的方法来获得事件源的相关信息。KeyEvent 类有许多比较常用的方法，如下所示。

- getSource()：返回触发此次事件的事件源组件对象，返回值类型为 Object。
- getKeyChar()：用来获得与引事件中的键相关联的字符。
- getKeyCode()：用来获得与此事件中的键相关联的整数 keyCode。
- getKeyText(int keyCode)：用来获得描述 keyCode 的文本，如“A”、“F1”、“HOME”等。
- isActionKey()：用来查看此事件中的键是否为“动作”键。
- isControlDown()：用来查看“Ctrl”键在此次事件中是否被按下，如果被按下，返回 true。
- isAltDown()：用来查看“Alt”键在此次事件中是否被按下，如果被按下，返回 true。
- isShiftDown()：用来查看“Shift”键在此次事件中是否被按下，如果被按下，返回 true。



在 KeyEvent 类中以“VK_”开头的静态常量代表各个按键的 keyCode，可以通过这些静态常量判断事件中的按键，以及获得按键的标签。

键盘事件的监听器类必须实现 KeyListener 接口。在 KeyListener 接口中有两个事件处理方法：focusGained()和 focusLost()，分别用来处理组件获得焦点事件和失去焦点事件。所以，实现了 KeyListener 接口的监听器类，必须实现 focusGained()方法和 focusLost()方法。KeyListener 接口的定义如下。

```
public interface KeyListener extends EventListener{
    public void keyTyped(KeyEvent e);       //当击键时将触发该方法
    publid void keyPressed(KeyEvent e);     //当键被按下时将触发该方法
    public void keyReleased(KeyEvent e);    //当释放按下的键时将触发该方法
}
```

为产生事件的组件注册键盘事件监听器，需要用到组件的 addFocusListener (KeyListener listener)方法。

17.8.3　处理键盘击键时所触发的事件

在键盘上击键时，会触发相应的键盘事件，这在程序设计当中经常用到。例如大家比较熟悉的俄罗斯方块游戏、贪吃蛇游戏、CS 游戏等，都是通过键盘上特定的键操作游戏的。下面的示例程序演示了键盘事件的使用。

【例 17.7】编写一个用来演示由键盘击键触发键盘事件的示例。程序的功能如图 17.15 所示。

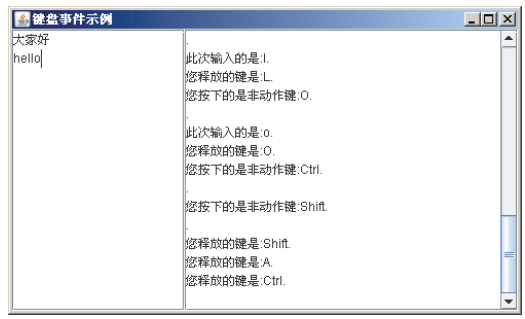


图 17.15　响应键盘事件

程序功能要求：

- 程序启动后，用户在左边文本域中输入任意字符，如图 17.15 所示。
- 在右边的文本域中显示击键信息。

实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter17，并在 chapter17 目录下，创建 java 源文件 KeyEventDemo.java；

② 使用记事本，打开 KeyEventDemo.java 源文件进行编辑，输入以下代码：

```
package chapter17;      //创建 chapter17 包
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



程序放在包 chapter17 中；需要导入 java.awt 包、java.awt.event 包、javax.swing 包。

③ 自定义文本域类 TextAreaListener。这个类还实现了键盘事件监听器接口 KeyListener。代码如下，所以它既是一个文本域，同时又是一个监听器。

```
//实现监听器类 TextAreaListener
class TextAreaListener extends JTextArea implements KeyListener{
    //构造方法，两个参数代表行和列宽
```

```
    public TextAreaListener(int row,int col){
        super(row,col);
        this.setEditable(false);    //将该文本框设为不可编辑，只用来显示击键信息
    }
    //实现 KeyListener 接口中的方法，当击键事件发生时，调用此方法处理击键事件
    public void keyTyped(KeyEvent e){
        this.append("此次输入的是:" + e.getKeyChar() + ".\n");
                                    //在文本框中追加所击键上的字符
    }
    //当键被按下时，调用此方法处理
    public void keyPressed(KeyEvent e){
        String keyText = KeyEvent.getKeyText(e.getKeyCode());
                                    //获得所击键的文本标签
        if(e.isActionKey())
            this.append("您按下的是动作键:" + keyText + ".\n");
        else{
            this.append("您按下的是非动作键:" + keyText + ".\n");
            char keyChar = e.getKeyChar();
            String str = "";
            switch(keyChar){
                case KeyEvent.VK_CONTROL:str = "Ctrl 键被按下.";break;
                case KeyEvent.VK_ALT:str = "Alt 键被按下.";break;
                case KeyEvent.VK_SHIFT:str = "Shift 键被按下.";break;
            }
            this.append(str + ".\n");               //向文本域中追加字符串
        }
    }
    //当键被释放时，调用此方法处理
    public void keyReleased(KeyEvent e){
        String keyText = KeyEvent.getKeyText(e.getKeyCode());
        this.append("您释放的键是:" + keyText + ".\n");
    }
}
```

类 TextAreaListener 继承自 swing 的 JTextField 组件，并实现了 KeyListener 监听器接口，作为键盘事件的监听器对象。

④ 编写窗体类 KeyEventFrame，为窗体依次添加两个文本域，并为左边的文本域添加监听器。具体代码如下。

```
//编写窗体类 KeyEventFrame
class KeyEventFrame extends JFrame{
    KeyEventFrame(){
        super();
        setTitle("键盘事件示例");
        setBounds(100,100,500,300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        TextAreaListener textAreaListener = new TextAreaListener(8,15);
//创建监听器对象及文本域
        JTextArea txtArea = new JTextArea(8,15);        //创建另一个文本域
        JScrollPane scrollPane1 = new JScrollPane(txtArea);
        JScrollPane scrollPane2 = new JScrollPane(textAreaListener);
        txtArea.addKeyListener(textAreaListener);       //添加事件监听器
        this.getContentPane().add(scrollPane1,BorderLayout.LINE_START);
```

```
                                                    //向内容面板中添加组件
        this.getContentPane().add(scrollPane2,BorderLayout.CENTER);
        this.setVisible(true);                      //显示窗体
    }
}
```

在程序窗体类 KeyEventFrame 的构造器中完成组件对象的初始化和界面布局。

⑤ 创建主程序 KeyEventDemo 类，具体代码如下。

```
public class KeyEventDemo{
    public static void main(String[] args){
        KeyEventFrame keyEventFrame=new KeyEventFrame();
    }
}
```

⑥ 使用下面的命令编译并运行主程序 KeyEventDemo.java。

```
javac -d . KeyEventDemo.java
java chapter17. KeyEventDemo
```

这时会看到如图 17.15 所示的程序界面。在左边的文本域中输入任意的内容。随着敲击键盘，会在右边的文本域中看到相应的击键信息。

17.9　处理鼠标事件

鼠标事件指的是当用户单击鼠标、鼠标进入某个组件区域、鼠标离开某个组件区域时，所触发的事件。任何组件都可以产生这些事件。

17.9.1　编写鼠标事件处理程序的步骤

要使程序能够处理鼠标事件，必须在程序中完成三个步骤：

① 创建一个鼠标事件监听器类。鼠标事件监听器类为实现了 MouseListener 接口的 Java 普通类。

② 实现鼠标事件监听器接口中的一个或一组事件处理方法。MouseListener 接口中，有五个事件处理方法：mouseEntered()、mousePressed()、mouseReleased()、mouseClicked() 和 mouseExited()。它们分别用来处理鼠标进入组件事件、鼠标被按下、鼠标被释放事件、单击鼠标事件和鼠标离开组件事件。所以实现了 MouseListener 接口的事件监听器类，必须实现这几个事件处理方法。当然，一般情况下，只需要其中一个或几个方法，将其他方法实现为空代码即可。

③ 为产生鼠标事件的组件注册这个事件监听器。通过调用组件的事件监听器添加方法：addMouseListener（MouseListener），向一个事件源组件注册鼠标事件监听器。

17.9.2　处理鼠标事件的过程

鼠标事件由 java.awt.event.MouseEvent 类捕获并封装事件的详细信息。当鼠标事件被触发时，封装的事件信息（MouseEvent 对象）会传递给已经注册的鼠标事件监听器，在监听器中就可以使用 MouseEvent 对象的方法来获得事件源的相关信息。MouseEvent 类有以下比较常用的方法。

- getSource()：返回触发此次事件的事件源组件对象，返回值类型为 Object。
- getButton()：用来获得触发此事件的鼠标按键，其返回值为 int 类型，对应 MouseEvent 类中定义的三个静态常量：MouseEvent.BUTTON1（代表鼠标左键）、MouseEvent.BUTTON2(代表鼠标滚轮)和 MouseEvent.BUTTON3(代表鼠标右键)。
- getClickCount()：用来获得鼠标被单击的次数，其返回值为一个整数。
- getPoint()：用来获得在该组件内鼠标被单击地方的 *x* 和 *y* 坐标，其返回值为一个 Point 对象。
- getX()：返回鼠标单击时的 *x* 位置。
- getY()：返回鼠标单击时的 *y* 位置。

鼠标事件的监听器类必须实现 MouseListener 接口。在 MouseListener 接口中有 5 个事件处理方法：分别用来处理相应原鼠标事件。所以，实现了 MouseListener 接口的监听器类，必须实现这 5 个方法。MouseListener 接口的定义如下。

```
public interface MouseListener extends EventListener{
    public void mouseEntered (MouseEvent e); //当鼠标移入组件时将触发该方法
    public void mousePressed (MouseEvent e); //当鼠标按键被按下时将触发该方法
    public void mouseReleased (MouseEvent e); //当鼠标按键被释放时将触发该方法
    public void mouseClicked (MouseEvent e); //当发生鼠标单击事件时将触发该方法
    public void mouseExited (MouseEvent e);  //当鼠标移出组件时将触发该方法
}
```

为产生事件的组件注册鼠标事件监听器，需要用到组件的 addMouseListener（MouseListener listener）方法。

17.9.3 处理鼠标操作所触发的事件

在使用鼠标进行交互时，会触发相应的鼠标事件。鼠标事件在程序设计当中也经常使用，例如使用鼠标单击按钮响应、使用鼠标玩射击游戏等。下面的示例程序演示了鼠标事件的处理过程。

【例 17.8】编写一个用来演示鼠标事件的示例。程序的功能如图 17.16 所示。

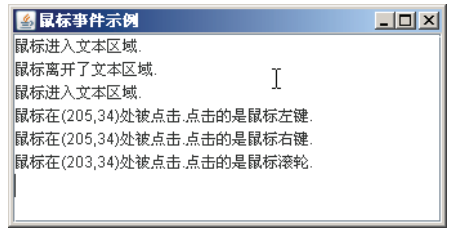


图 17.16　响应鼠标事件

程序功能要求：

- 程序启动后，将鼠标移入文本域，如图 17.16 所示。
- 将鼠标放在窗体标题栏上，按下鼠标左键并拖动窗体到合适的位置。
- 在文本域中左击、右击鼠标键。在文本域中显示相应的信息。
- 在文本域中单击中间的鼠标滚轮。在文本域中显示相应的信息。

在做以上操作时，注意观察文本域中的信息。实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter17，并在 chapter17 目录下，创建 java 源文件 MouseEventDemo.java；

② 使用记事本，打开 MouseEventDemo.java 源文件进行编辑，输入以下代码：

```
package chapter17;      //创建 chapter17 包
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



程序放在包 chapter17 中；需要导入 java.awt 包、java.awt.event 包、javax.swing 包。

③ 编写窗体类 KeyEventFrame，并将其实现为一个鼠标事件监听器。为窗体添加一个文本域，并将当前窗体注册为文本域对象的监听器。具体代码如下。

```
//编写窗体类 MouseEventFrame
class MouseEventFrame extends JFrame implements MouseListener{
    JTextArea txtArea;                          //声明一个文本域对象
    public MouseEventFrame(){
        super();
        setTitle("鼠标事件示例");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        txtArea = new JTextArea(8,30);          //创建文本域对象
        JScrollPane scrollPane = new JScrollPane(txtArea);
        txtArea.addMouseListener(this);         //将当前窗体注册为文本域 txtArea
的鼠标事件监听器
        this.getContentPane().add(scrollPane,BorderLayout.CENTER);
        this.pack();
        this.setVisible(true);
    }
    //当鼠标移入组件时将触发该方法
    public void mouseEntered (MouseEvent e){
        txtArea.append("鼠标进入文本区域.\n");
        txtArea.setCaretPosition(txtArea.getText().length());    //设置光标
置于文本结尾处
    }
    //当鼠标按键被按下时将触发该方法,在这里不作处理
    public void mousePressed (MouseEvent e){}
    //当鼠标按键被释放时将触发该方法,在这里不作处理
    public void mouseReleased (MouseEvent e){}
    //当发生鼠标单击事件时将触发该方法
    public void mouseClicked (MouseEvent e){
        txtArea.append("鼠标在(" + e.getX() + "," + e.getY() + ")处被单击.");
        int i = e.getButton();
        String str = "";
        //判断用户所单击的鼠标键
        switch(i){
            case MouseEvent.BUTTON1:            //如果单击的是鼠标左键
                str = "单击的是鼠标左键.\n";
                break;
            case MouseEvent.BUTTON3:            //如果单击的是鼠标右键
                str = "单击的是鼠标右键.\n";
                break;
            case MouseEvent.BUTTON2:            //如果单击的是鼠标中间键或鼠标滚轮
                str = "单击的是鼠标滚轮.\n";
        }
        txtArea.append(str);                    //向文本区中添加字符串
```

```
            txtArea.setCaretPosition(txtArea.getText().length());
                                                    //设置光标置于文本结尾处
        }
        //当鼠标移出组件时将触发该方法
        public void mouseExited (MouseEvent e){
            txtArea.append("鼠标离开了文本区域.\n");
            txtArea.setCaretPosition(txtArea.getText().length());
        }
    }
```

将窗体类 MouseEventFrame 实现为鼠标事件监听器，因此在 MouseEventFrame 类中必须实现鼠标事件监听器中的所有鼠标事件处理方法。对于不需要的事件处理方法，方法体为空即可。

④ 创建主程序 MouseEventDemo 类，具体代码如下。

```
public class MouseEventDemo{
    public static void main(String[] args){
        new MouseEventFrame();
    }
}
```

⑤ 使用下面的命令编译并运行主程序 KeyEventDemo.java。

```
javac -d . MouseEventDemo.java
java chapter17. MouseEventDemo
```

这时会看到如图 17.16 所示的程序界面。将鼠标试着进入文本域，单击鼠标，移动鼠标滚轮，离开文本域，观察在文本域中显示的事件信息。

17.10 处理鼠标移动事件

鼠标移动事件发生在鼠标移过某个组件时。与鼠标事件一样，任何组件都可以产生鼠标移动事件。处理鼠标移动事件需要使用 MouseMotionListener 接口。

17.10.1 编写鼠标移动事件处理程序的步骤

要使程序能够处理鼠标移动事件，必须在程序中完成三个步骤：

① 创建一个鼠标移动事件监听器类。鼠标移动事件监听器类为实现了 MouseMotionListener 接口的 Java 普通类。

② 实现鼠标移动事件监听器接口中的一个或一组事件处理方法。MouseMotionListener 接口中，有两个事件处理方法：mouseDragged()和 mouseMoved()。它们分别用来处理鼠标拖动和鼠标移动事件。所以实现了 MouseMotionListener 接口的事件监听器类，必须实现这两个事件处理方法。

③ 为产生鼠标移动事件的组件注册这个事件监听器。通过调用组件的事件监听器添加方法：addMouseMotionListener（MouseMotionListener listener），向一个事件源组件注册鼠标移动事件监听器。

17.10.2 处理鼠标移动事件的过程

鼠标移动事件由 java.awt.event. MouseEvent 类捕获并封装事件的详细信息。当鼠标移



动事件被触发时，封装的事件信息（MouseEvent 对象）会传递给已经注册的鼠标移动事件监听器，在监听器中就可以使用 MouseEvent 对象的方法来获得事件源的相关信息。MouseEvent 类有以下比较常用的方法。

- getSource()：返回触发此次事件的事件源组件对象，返回值类型为 Object。
- getButton()：用来获得触发此事件的鼠标按键，其返回值为 int 类型，对应 MouseEvent 类中定义的三个静态常量：MouseEvent.BUTTON1（代表鼠标左键）、MouseEvent.BUTTON2（代表鼠标滚轮）和 MouseEvent.BUTTON3（代表鼠标右键）。
- getClickCount()：用来获得鼠标被单击的次数，其返回值为一个整数。
- getPoint()：用来获得在该组件内鼠标被单击地方的 *x* 和 *y* 坐标，其返回值为一个 Point 对象。
- getX()：返回鼠标单击时的 *x* 位置。
- getY()：返回鼠标单击时的 *y* 位置。

鼠标移动事件的监听器类必须实现 MouseMotionListener 接口。在 MouseMotionListener 接口中有两个事件处理方法：分别用来处理相应原鼠标移动事件。所以，实现了 MouseMotionListener 接口的监听器类，必须实现这两个方法。MouseMotionListener 接口的定义如下。

```
public interface MouseMotionListener extends EventListener{
    public void mouseDragged (MouseEvent e); //当在组件中拖动鼠标时将触发该方法
    public void mouseMoved (MouseEvent e);   //当在组件中移动鼠标时将触发该方法
}
```

为产生事件的组件注册鼠标移动事件监听器，需要用到组件的 addMouseMotionListener（MouseMotionListener listener）方法。

17.10.3　处理鼠标移动时所触发的事件

在组件中移动鼠标时，会触发相应的鼠标移动事件。在使用鼠标移动时，会触发相应的鼠标移动事件。鼠标移动事件在程序设计当中也经常使用，例如使用鼠标进行拖曳、使用鼠标进行绘图等。下面的示例程序演示了鼠标移动事件的处理过程。

【例 17.9】编写一个用来演示鼠标移动事件的示例。程序的功能如图 17.17 所示。

程序功能要求：

- 程序启动后，将鼠标移入文本域，如图 17.17 所示。
- 在文本域内移动鼠标，在文本域中显示当前鼠标所在的坐标。

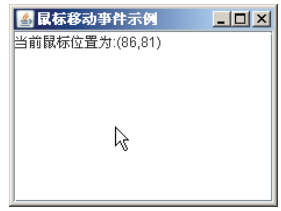


图 17.17　响应鼠标事件

在做以上操作时，注意观察文本域中的信息。实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter17，并在 chapter17 目录下，创建 java 源文件 MouseMotionEventDemo.java；

② 使用记事本，打开 MouseMotionEventDemo.java 源文件进行编辑，输入以下代码：

```
package chapter17;      //创建 chapter17 包
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
```



程序放在包 chapter17 中；需要导入 java.awt 包、java.awt.event 包、javax.swing 包。

③ 编写自定义的文本域 MouseMotionTextArea，它派生自 JTextArea 类，并将其实现为一个鼠标移动事件的监听器。代码如下。

```
//自定义文本域类，并将其实现为鼠标移动事件监听器
class MouseMotionTextArea extends JTextArea implements MouseMotionListener{
    public MouseMotionTextArea(int row,int col){
        super(row,col);
        this.setEditable(false);
        this.addMouseMotionListener(this); //将自身注册为鼠标移动事件的监听器
    }
    //当在组件中拖动鼠标时将触发该方法
    public void mouseDragged (MouseEvent e){}
    //当在组件中移动鼠标时将触发该方法
    public void mouseMoved (MouseEvent e){
        setText("");                          //清空文本域中的文本
        Point point = e.getPoint();           //获得当前鼠标所在的坐标
        setText("当前鼠标位置为:(" + point.x + "," + point.y + ")");
    }
}
```

自定义类 MouseMotionTextArea 继承自 swing 组件 JTextArea，并实现了鼠标移动事件监听器接口，由它来监听并处理鼠标移动事件。

④ 编写窗体类 MouseMotionEventFrame，并为窗体添加一个文本域（MouseMotionTextArea 对象），并将当前窗体注册为文本域对象的监听器。具体代码如下。

```
//编写窗体类 MouseEventFrame
class MouseMotionEventFrame extends JFrame{
   public MouseMotionEventFrame(){              //构造器
        super();                                //调用父类构造器
        setTitle("鼠标移动事件示例");             //设置窗体标题
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //设置当关闭窗体时，
退出程序
        MouseMotionTextArea txtArea = new MouseMotionTextArea(8,20);
//创建自定义文本区并指定大小
        JScrollPane scrollPane = new JScrollPane(txtArea); //创建包含自定
义文本区的滚动面板
        this.getContentPane().add(scrollPane,BorderLayout.CENTER);
//将滚动面板添加到内容面板中
        this.pack();
        this.setVisible(true);                  //显示窗体
   }
}
```

在窗体类 MouseMotionEventFrame 的构造器中添加组件并完成界面布局。

⑤ 创建主程序 MouseMotionEventDemo 类，具体代码如下。

```
//编写主程序
public class MouseMotionEventDemo{
```

```
    public static void main(String[] args) {
        MouseMotionEventFrame f = new MouseMotionEventFrame();
    }
}
```

⑥ 使用下面的命令编译并运行主程序 MouseMotionEventDemo.java。

```
javac -d . MouseMotionEventDemo.java
java chapter17. MouseMotionEventDemo
```

这时会看到如图 17.17 所示程序界面。在程序的文本域中移动鼠标，在文本域中会显示鼠标所在的坐标位置。随着鼠标的移动，显示的坐标值也在不断发生变化。如果将鼠标移出文本域外，这时文本域组件将不会再发生鼠标移动事件，因此坐标值就不再发生变化。

17.11　实例：一个简单的用户登录验证程序

在第 16 章，已经学习布局了一个简单的用户登录验证程序。本节继续完善此程序，补充完整功能代码，使其能够具有登录验证功能。程序运行初始界面如图 17.18 所示。

17.11.1　程序功能

用户登录验证是大多数商业程序都会具有的一个功能模块。通过对登录用户的账户和密码进行验证，可以有效地控制对相关信息的访问，或者限定访问权限。在登录验证时，会响应许多的用户事件。

【例 17.10】实现一个简单的用户登录验证程序的示例。程序功能要求：

- 程序启动后，不输入账户和密码，直接单击【确定】按钮，弹出【警告】对话框，如图 17.19 所示。



图 17.18　用户登录验证程序界面

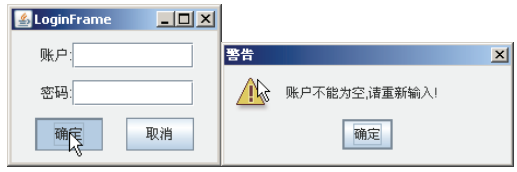


图 17.19　账户为空，单击【确定】按钮

- 单击对话框【确定】按钮，光标自动定位于【账户】文本框。
- 输入账户但不输入密码，单击【确定】按钮，弹出【警告】对话框，如图 17.20 所示。
- 单击对话框【确定】按钮，光标自动定位于密码框内。
- 输入账户和错误密码，单击【确定】按钮，弹出【警告】对话框，显示密码不正确，要求重新输入密码，如图 17.21 所示。

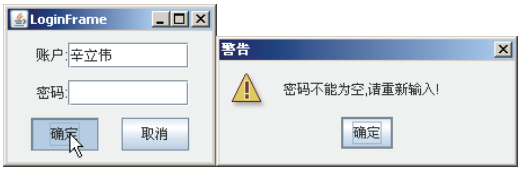


图 17.20　密码为空，单击【确定】按钮



图 17.21　密码不正确，单击【确定】按钮

- 单击对话框【确定】按钮，光标自动定位于密码框内，并清空密码框。

- 输入账户和正确密码“123456”，单击【确定】按钮，弹出【成功登录】提示对话框，如图 17.22 所示。

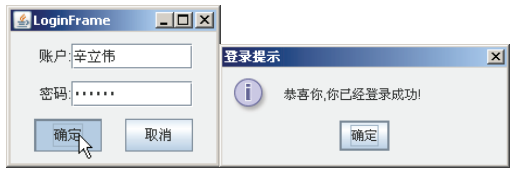


图 17.22　密码正确，单击【确定】按钮

- 单击【成功登录】提示对话框中的【确定】按钮，结束并退出程序。

在这里，只对用户输入的密码正确与否做了验证，而账户可以任意输入内容，但不能为空。

17.11.2　为程序实现动作事件监听器

将原来的代码做如下更改，在类 LoginFrame 的声明中，增加“implements ActionListener”声明，实现 ActionListener 监听器，如下所示。

```
package chapter17;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class LoginFrame implements ActionListener{     //声明类LoginFrame
为实现了 ActionListener 的监听器
    JButton buttonOk,buttonCancel;                      //创建按钮对象
    JTextField textFieldName;                           //创建文本框对象
    JPasswordField textFieldPwd;                        //创建密码框对象
    ……
}
```

其中，导入 java.awt.event.*包，因为动作事件监听器类及动作事件类在此包中定义。并且因为要在事件处理方法中设置按钮的状态，所以将按钮 buttonOk 和 buttonCancel、文本框 textFieldName 和密码框 textFieldPwd 设为实例变量。将类 LoginFrame 实现为动作监听器，实现 ActionListener 接口。

17.11.3　为按钮注册监听器

在方法 addComponentsToPane 中，当创建按钮 buttonOk 和 buttonCancel 的实例以后，分别为它们注册监听器，代码如下。

```
public void addComponentsToPane(Container pane) {
    ……
    buttonOk = new JButton("确定");
    buttonOk.addActionListener(this);       //将 LoginFrame 注册为“确定”按钮
的动作事件监听器
    buttonCancel = new JButton("取消");
    buttonCancel.addActionListener(this);   //将 LoginFrame 注册为“取消”按钮
的动作事件监听器
    ……
```



```
    }
```

17.11.4　实现 ActionListener 接口中的 actionPerformed 方法

接下来，实现 ActionListener 接口中的 actionPerformed 方法，代码如下。

```
    //实现 ActionListener 接口中的动作事件处理方法
    public void actionPerformed(ActionEvent e){
        JButton button = (JButton)e.getSource();   //获得事件源按钮
        if(button == buttonOk){     //如果是“确定”按钮
            checkAccount();         //调用检查登录账户和密码的方法对账户进行检查
        }else{
            resetAccount();         //调用重置登录账户和密码的方法
        }
    }
```

其中 checkAccount 和 resetAccount 是完成相应功能的自定义方法，其定义如下。

```
    //检查用户账户是否为空.如果为空,则将光标定位在账户文本框中
    public void checkAccount(){
        String userName = textFieldName.getText().trim();  //获取用户输入的账户
        char[] pwdChars = textFieldPwd.getPassword();     //获取用户输入的密码,
为字符数组
        String password = new String(pwdChars).trim(); //将用户密码构建为字符串
        if(userName.length() == 0){                    //如果用户输入的账户为空
            //弹出警告信息对话框
            JOptionPane.showMessageDialog(null,"账户不能为空,请重新输入!",
                                "警告",JOptionPane.WARNING_MESSAGE);
            textFieldName.requestFocusInWindow();      //强制账户文本框获得焦点
        }else if(password.length() == 0){              //如果用户输入的密码为空
            //弹出警告信息对话框
            JOptionPane.showMessageDialog(null,"密码不能为空,请重新输入!",
                            "警告",JOptionPane.WARNING_MESSAGE);
            textFieldPwd.requestFocusInWindow();       //强制密码框获得焦点
        }else if(!password.equals("123456")){          //如果用户输入的密码不是
"123456"
            //弹出警告信息对话框
            JOptionPane.showMessageDialog(null,"密码不正确,请重新输入!",
                            "警告",JOptionPane.WARNING_MESSAGE);
            textFieldPwd.setText("");                  //清空旧密码
            textFieldPwd.requestFocusInWindow();       //重新获得输入焦点
        }else{
            //弹出成功登录信息对话框
            JOptionPane.showMessageDialog(null,"恭喜你,你已经登录成功!",
                            "登录提示",JOptionPane.INFORMATION_MESSAGE);
            System.exit(0);                            //结束程序
        }
    }
    //重置用户账户文本框和密码框
    public void resetAccount(){
        textFieldName.setText("");                     //清空用户账户文本框
        textFieldPwd.setText("");                  //清空密码框
        textFieldName.requestFocusInWindow();  //强制用户账户文本框获得输入焦点
```

```
}
```

在上述代码注释中，自定义了检查用户账户是否为空的 checkAccount()方法，以及重置用户账户文本框和密码框的 resetAccount()方法。当程序响应按钮单击操作时，会在事件处理方法 actionPerformed()中，根据用户单击的按钮类型，分别调用这两个方法进行相应的事件处理。

17.11.5 编译并运行程序

使用如下的命令编译并运行 LoginFrame.java 程序：

```
javac -d . LoginFrame.java
java chapter17. LoginFrame
```

至此，已经成功地编写了一个简单的用户登录验证程序。当然，作为一个示例，这个程序还相当简单。在实际开发中，账户和密码一般是不能硬编码在代码中的。用户输入的账户和密码往往需要和数据库中已注册的账户和密码进行比较判断。在这里，主要是用来给读者演示事件处理代码的编写过程，所以进行了简化。

17.12 小结

本章介绍了程序事件处理原理和 Java 中的事件处理机制，并且概括了 Java 中的各种事件类型。在学习本章内容时，重点掌握各种事件类型的处理方法。在了解 Java 的事件处理机制时，可能会有一定的难度。下一章介绍 Java 程序中的图形、文本和图像处理。

第 18 章　图形、文本和图像处理

Java 语言得以流行的原因之一，就是它成功地应用于网络，并使网页由静态变为可以带有声音、图像、图形及动画的动态网页，使互联网一下子丰富和精彩起来。虽然现在大多数的 Java 开发是基于 Web 的，人们似乎更关注 Java 企业级开发方面的表现（J2EE），但实际上，Java 在图形界面程序开发上，提供了同样强大的功能和丰富的 API 库，其中就包括图形、图像和字体的处理。

18.1　Java2D 概述

在 Java 类库中，对图形提供了强大的支持，通过简单的几条相关语句就可以得到复杂的视觉效果。而 Java2D 是在 Java2 中提出的类，它提供了一些增强的特性，通过对抽象窗口工具（AWT）的扩展，提供二维图形、图像和文本的绘制能力。

18.1.1　Java2D 坐标空间

要能在组件上绘制二维的文本或图形图像，必须知道要绘制对象的坐标才行。例如要画一条直线，那么要有一个（*x*,*y*）坐标来标识它的起点，还要有另一个（*x*,*y*）坐标来标识它的终点。在 Java 2D 的 API 中，维持有两种坐标空间：

- 用户空间：指的是在程序中指定原始图形时所使用的空间。
- 设备空间：指的是输出设备如屏幕、窗口或打印机的坐标系统。

用户空间是一个独立于硬件设备的逻辑坐标系统，它是由程序使用的坐标空间。在编写程序时，所指定的坐标都属于用户空间坐标系统。

设备空间指的是硬件设备所使用的坐标空间。因此，将程序中指定的图形、图像或文本输出到硬件设备（如打印机、屏幕等）上时，要将用户坐标空间转换为设备坐标空间。在默认情况下，用户空间的坐标原点和设备空间的坐标原点是相同的，都位于组件绘制区域的左上角。其中 *x* 坐标轴向右增长，*y* 坐标轴向下增长，如图 18.1 所示。

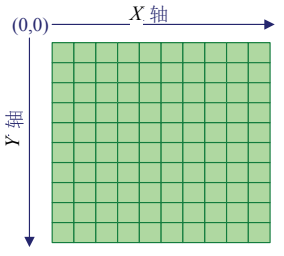


图 18.1　Java 图形化坐标系统

窗口的左上角的坐标为（0，0）。通常情况下，所有的坐

标都使用整数来指定。不过有时也要求使用浮点数或 double 类型的精度坐标，这也支持。

设备空间是一个依赖于设备的坐标系统，它根据目标设备的不同而不同。虽然用于窗口或屏幕的坐标系统与用于打印机的坐标系统很不相同，但对于 Java 程序来说这些不同是透明的。在图形或图像渲染的时候，用户空间和设备空间之间的必要转换为自动执行。

18.1.2 Java2D 渲染模型

Java 2D 为跨不同类型的设备提供了统一的渲染模型。当一个组件需要被显示时，其 paint 或 update 方法被相应的 Graphics 上下文环境自动地调用。可以将 Graphics（包括 Graphics2D）理解为绘图的上下文环境，例如，如果在打印机上绘图，那么它就代表打印机的绘图设备环境；如果在内存中绘图，它就代表内存中的绘图环境。

在 Java 2D 中，包括一个 java.awt.Graphics2D 的类，继承自 Graphics 类，提供了对 Java 2D 的一些增强的图形和渲染的访问。这些特性包括：

- 对几何图形轮廓的渲染，如可以绘制不同宽度和样式的直线（使用 draw()方法）。
- 对任何几何图形使用指定的颜色和模式进行内部填充（如渐近填充模式）的渲染（fill()方法）。
- 对任何文本字符串的渲染（drawString()方法）。
- 对指定的图像进行渲染（drawImage()方法）。

此外，Graphics2D 类也支持 Graphics 的一些特定的渲染方法，如 drawLine()、drawOval()和 fillRect()。所有这些绘图渲染方法大致可分为两类：一类是用来绘制图形的方法，另一类是设置绘制属性（渲染特性）的方法。

在 Java 中，绘图操作都是在 Graphics 对象或 Graphics2D 对象上调用的。选择到底使用 Graphics 对象还是 Graphics2.0 对象，这取决于是否使用 Java2D 图形特征。所有的 Java2D 图形操作都必须在一个 Graphics2D 对象上调用。

 18.2 Java2D 绘图入门

在 Java 的应用程序中，不能直接绘图，通常是在一个空白面板或者画布(Canvas)上进行绘图。所以在绘图前要先创建一个绘图面板或者画布。

18.2.1 创建一个绘图面板

绘图操作必须在一个组件表面上进行。一般最适合用来绘图的组件就是 javax.swing.JPanel。这个类代表了在图形化用户界面中的面板。例如，下面的代码创建一个窗体和一个面板，并将该面板加入到窗体的内容空格中。

```
JFrame frame = new JFrame("绘图程序");
JPanel pane = new JPanel();
frame.getContentPane.add(pane);
```

JPanel 和其他 swing 组件一样，它有一个 paintComponent(Graphics)方法。每当该组件应该被重新显示时，它会被自动调用，以重新绘制自己。通过创建一个 JPanel 的子类，重写这个 paintComponent()方法，并将所有的绘图操作放在这个方法里，那么当组件重绘时，



就会执行自定义的绘图操作。

18.2.2　获取 Graphics2D 对象

Graphics2D 类的对象代表一个可以在里边绘制某些内容的环境，它可以代表一个面板、一个组件的表面或者屏幕、打印机。执行具有 Java2D 图形特性的绘图操作必须在 Graphics2D 对象上调用相应的绘图方法。所有 Graphics2D 也称为“绘图上下文”。

在开始绘图之前，必须首先获取一个 Graphics2D 对象，如以下代码所示。

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;
}
```

要获得一个 Graphics2D，必须通过这种强制转换将一个 Graphics 对象转换为 Graphics2D 对象，然后就可以调用这个 Graphics2D 对象上的各种绘图方法了。

18.2.3　简单实例：绘制字符串

获得 Graphics2D 以后，就可以调用其包含的各种方法进行绘图操作。其中最常用的一个方法是用来绘制字符串的 drawString()方法。

【例 18.1】编写一个应用程序的示例。程序的功能是在面板上绘制“大家好，这是我的第一个绘图程序”字符串，如图 18.2 所示。

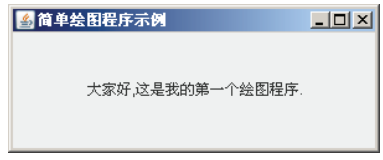


图 18.2　简单绘图示例

实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter18，并在 chapter18 目录下，创建 java 源文件 SimpleDrawDemo.java；

② 使用记事本，打开 SimpleDrawDemo.java 源文件进行编辑，输入以下代码：

```
package chapter18;      //创建 chapter18 包
import java.awt.*;
import javax.swing.*;
```



程序放在包 chapter18 中；需要导入 java.awt 包和 javax.swing 包。

③ 接着编写自定义的 JPanel 类，并重写其 paintComponent 方法，将绘制字符串的方法放在这个方法中。代码如下。

```
//自定义一个 JPanel 类,重写其 paintComponent 方法
class MyPane extends JPanel{
   public void paintComponent(Graphics g){
        Graphics2D g2 = (Graphics2D)g; //强制转换获得一个 Graphics2D 对象
        g2.drawString("大家好,这是我的第一个绘图程序.",60,50); //绘制字符串
    }
}
```

绘图操作必须放在 paintComponent()方法中，用来重绘组件表面。

④ 编写程序窗体界面类 SimpleDrawFrame，该类继承自 JFrame 类。在该类中添加了一个自定义面板对象 myPane。具体代码如下。

```
//编写应用程序 SimpleDrawFrame
```

```
class SimpleDrawFrame extends JFrame{
    SimpleDrawFrame(){
        super();
        setTitle("简单绘图程序示例");                          //设置窗体标题
        setBounds(100,100,300,120);                          //设置窗体位置和大小
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置关闭模式
        MyPane myPane = new MyPane();          //创建自定义面板的对象实例
        this.getContentPane().add(myPane,BorderLayout.CENTER);
        this.setVisible(true);                               //显示窗体
    }
}
```

在程序窗体界面类 SimpleDrawFrame 的构造器中完成窗体布局。

⑤ 创建主程序 SimpleDrawDemo，具体代码如下。

```
//编写主程序
public class SimpleDrawDemo{
    public static void main(String[] args){
        new SimpleDrawFrame();              //创建主程序类对象，运行程序
    }
}
```

⑥ 使用下面的命令编译并运行主程序 SimpleDrawDemo.java。

```
javac -d . SimpleDrawDemo.java
java chapter18. SimpleDrawDemo
```

编译并运行以后，就会看到如图 18.2 所示的程序运行界面。可以看到，在容器面板中已经有了绘制的字符串。试着将该窗体最小化、最大化和移动，这些操作都会引起窗体及其中的组件重绘，包括面板中的字符串。

18.3 绘制几何图形

在掌握了 Java2D 绘图的基本知识后，如坐标空间、获得和使用 Graphics2D 对象及简单的绘图操作等，本节开始学习更加详细的 Java2D 的绘图特性。将学习如何绘制几何图形和任何其他的图形，如何填充这些图形，如何设置绘制和填充特性等。

18.3.1 基本几何对象

在 Java 中，每一个基本的几何类都有相对应的类，这些几何类都位于 java.awt.geom 包中。使用 Java 提供的几何类，可以大大减轻程序员处理几何图形时的工作量，提高开发效率。Java 所提供的几何类如表 18.1 所示。

表 18.1 基本几何类

几何图形	对应类名	实现子类
点	Point2D	Point2D.Float 和 Point2D.Double
直线	Line2D	Line2D.Float 和 Line2D.Double
矩形	Rectangle2D	Rectangle2D.Float 和 Rectangle2D.Double
圆角矩形	RoundRectangle2D	RoundRectangle2D.Float 和 RoundRectangle2D.Double
椭圆	Ellipse2D	Ellipse2D.Float 和 Ellipse2D.Double



续表

几何图形	对应类名	实现子类
弧形	Arc2D	Arc2D.Float 和 Arc2D.Double
二次曲线	QuanCurve2D	QuardCurve2D.Float 和 QuardCurve2D.Double
三次曲线	CubicCurve2D	CubicCurve2D.Float 和 CubicCurve2D.Double
任意图形	GeneralPath	--
区域（面积）	Area	--

要绘制相应的几何图形，首先要创建这些几何类的对象实例，然后获得一个 Graphics2D 对象，并调用 Graphics2D 对象的方法来绘制这些几何图形。

18.3.2　使用 Point2D

使用 Point2D.Float 或 Point2D.Double 创建一个点对象，一个点对象代表坐标空间中(*x*,*y*)位置。可以使用如下的方式创建一个坐标为（45,40）的点对象。

```
Point2D.Double point1 = new Point2D.Double(45,40);
```

也可以使用不带参数的构造器构造一个点对象，然后使用 setLocation()方法设置其位置，如以下代码所示。

```
Point2D.Double point2 = new Point2D.Double();
point2.setLocation(45,50);
```

或者，使用已知点的坐标设置当前点的坐标，两个点有相同的坐标位置。这需要使用带有一个 Point 类型对象的参数的构造器来实现，如下所示。

```
point2.setLocation(point1);
```

另外，Point2D 对象还提供有计算两个点之间距离的方法 distance()，如下所示。

```
double distance = point1.distance(point2); //计算 point1 和 point2 两点之间的距离
```

可以重写面板对象的 paintComponent()方法，在该方法中，创建两个点，然后计算这两个点之间的距离。例如：

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;          //强制转换一个 Graphics2D 对象
    Point2D.Double point1 = new Point2D.Double(45,40);     //创建点 point1
    Point2D.Double point2 = new Point2D.Double();  //创建点 point2
    point2.setLocation(45,50);                          //设置点 point2 的坐标
    double distance = point1.distance(point2);      //计算两个点之间的距离
    String str = "(45,40)和(45,50)两点之间的距离为" + distance;
    g2.drawString(str,30,40);                  //在组件上绘制相应的字符串
}
```

正如上一节所述，绘图代码必须放在 paintComponent()方法中。

18.3.3　画直线

在 Java2D 中，要画一条直线，首先要使用 Line2D.Float 对象或 Line2D.Double 对象创建这个直线片段对象。其语法格式如下。

```
Line2D.Double(double x1,double y1,double x2,double y2);
Line2D.Double(Point2D p1,Point2D p2);
```

```
Line2D.Float(float x1, float y1, float x2, float y2));
Line2D.Float(Point2D p1,Point2D p2));
```

其中前两个参数为直线的起始点的坐标，后两个参数为直线的终点的坐标。例如可以使用如下的方式创建一个 double 精度的直线对象。

```
Line2D.Double line1 = new Line2D.Double(60,20,100,50);
```

也可以使用不带参数的构造器构造一个直线对象，然后使用 setLine()方法设置直线的端点，如以下代码所示。

```
Line2D.Double line2 = new Line2D.Double();
line2. setLine(60,60,100,60);
```

可以重写面板对象的 paintComponent()方法，在该方法中，绘制两条直线。例如：

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;          //强制转换一个 Graphics2D 对象
    Line2D.Double line1 = new Line2D.Double(60,20,100,50); //创建直线对象 line1
    Line2D.Double line2 = new Line2D.Double(); //创建另一条直线对象
    line2. setLine(60,60,100,60);           //设置直线 line2 的两个端点
    g2.draw(line1);                         //画直线 line1
    g2.draw(line2);                         //画直线 line2
    g2.drawString("画直线",60,90);          //画字符串
}
```

正如上一节所述，绘图代码必须放在 paintComponent()方法中。

18.3.4 画直角矩形

类 Rectangle2D 代表一个矩形。使用 Rectangle2D.Float 或 Rectangle2D.Double 创建一个 float 精度或 double 精度的直角矩形。其语法格式如下。

```
Rectangle2D.Double(double x,double y,double rectwidth,double rectheight);
Rectangle2D.Float(float x,float y,float rectwidth,float rectheight);
```

其中前两个参数为矩形的左上角的坐标，后两个参数分别为矩形的宽和高。例如可以使用如下的方式创建一个 double 精度的直角矩形对象。

```
Rectangle2D.Double rect = new Rectangle2D.Double(120,25,60,50);
```

可以重写面板对象的 paintComponent 方法，在该方法中，绘制一个直角矩形。例如：

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;          //强制转换一个 Graphics2D 对象
    Rectangle2D.Double rect = new Rectangle2D.Double(120,25,60,50);
                                            //创建直角矩形对象 rect
    g2.draw(rect);                          //画直角矩形
}
```

正如上一节所述，绘图代码必须放在 paintComponent 方法中。

18.3.5 画圆角矩形

类 Rectangle2D 代表一个矩形。使用 RoundRectangle2D.Float 或 RoundRectangle2D.Double 创建一个 float 精度或 double 精度的圆角矩形。其语法格式如下。

```
RoundRectangle2D.Double(double x,double y,                    //圆角矩形左上角坐标
                        double rectwidth,double rectheight,  //矩形的宽和高
                        double arcwidth,double archeight);
                                                    //圆角弧度所在矩形的宽和高
RoundRectangle2D.Float(float x, float y,          //圆角矩形左上角坐标
                        float rectwidth, float rectheight,   //矩形的宽和高
                        float arcwidth, float archeight);
                                                    //圆角弧度所在矩形的宽和高
```



图 18.3　圆角矩形

其中前两个参数为矩形的左上角的坐标，中间两个参数分别为矩形的宽和高，后两个参数为圆角弧所在矩形的宽和高。创建的椭圆与其限定的矩形的关系如图 18.3 所示。

例如可以使用如下的方式创建一个 double 精度的圆角矩形对象：

```
RoundRectangle2D.Double roundrect = new RoundRectangle2D.Double(200,25,60, 50,10,10);
```

可以重写面板对象的 paintComponent 方法（绘图代码必须放在 paintComponent 方法中）。在该方法中，绘制一个圆角矩形。例如：

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;          //强制转换一个 Graphics2D 对象
    //创建圆角矩形对象 roundrect
   RoundRectangle2D.Double roundrect = new RoundRectangle2D.Double(200,25,60,50,10,10);
    g2.draw(roundrect);                     //画圆角矩形
}
```

18.3.6　画椭圆形

类 Ellipse2D 代表一个椭圆。使用 Ellipse2D.Float 或 Ellipse2D.Double 创建一个 float 精度或 double 精度的椭圆形。其语法格式如下。

```
Ellipse2D.Double(double x,double y,double rectwidth,double rectheight);
Ellipse2D.Float(float x,float y,float rectwidth,float rectheight);
```

从语法上看，其参数与创建直角矩形的参数相同。其中前两个参数为矩形的左上角的坐标，后两个参数分别为矩形的宽和高，而所创建的椭圆的外边框为所设定的矩形。创建的椭圆与其限定的矩形的关系如图 18.4 所示。



图 18.4　椭圆与矩形的关系

例如可以使用如下的方式创建一个 double 精度的椭圆对象：

```
Ellipse2D.Double ellipse = new Ellipse2D.Double(50,125,60,50);
```

可以重写面板对象的 paintComponent 方法（绘图代码必须放在 paintComponent 方法中），在该方法中，绘制一个直角矩形。例如：

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;          //强制转换一个 Graphics2D 对象
    Ellipse2D.Double ellipse = new Ellipse2D.Double(50,125,60,50);
                                            //创建椭圆形对象
    g2.draw(ellipse);                       //画椭圆形
}
```

18.3.7 画弧形

画弧比较复杂。弧形是椭圆形的一部分，有一个的起始角度和弧所经过的度数。弧可以封闭也可以不封闭。在 Swing 中，使用 Arc2D 类来代表一个弧形。它有两个子类，Arc2D.Float 和 Arc2D.Double。Arc2D 类使用如下三个常量代表三种类型的弧：

- Arc2D.OPEN：该闭合类型针对不具有用来连接弧段两个端点的路径段的开弧。
- Arc2D.PIE：该闭合类型针对通过从弧段的起始点到完整椭圆的中心、再从圆心到弧段的结束点画直线来闭合的弧。
- Arc2D.CHORD：该闭合类型针对通过从弧段的起始点到弧段的结束点画一条直线来闭合的弧。

其语法格式如下。

```
Arc2D.Double(double x,double y,double rectwidth,double rectheight,
                                                    //所在矩形
                 double startangle,double angular,
                                                    //起始角度和弧逆时针经过的角度
                 int closuretype));                 //弧形样式
Arc2D.Double(float x, float y, float rectwidth, float rectheight,
                                                    //所在矩形
                 float start, float extent,//起始角度和弧逆时针经过的角度
                 int type));                        //弧形样式
```

弧形起始角度的范围是逆时针方向上的 0 度到 359 度。在一个环形椭圆上，0 度是 3 点钟的方向，90 度是 12 点钟的方向，180 度是 9 点钟的方向，而 270 度是 6 点钟的方向。

弧形所经过的度数是逆时针方向上的从 0 度到 359 度的范围，也可以是顺时针方向上的 0 度到-359 度。创建的弧形与其所属椭圆及限定的矩形的关系如图 18.5 所示。



图 18.5 弧、椭圆与矩形的关系

例如，下面的代码创建一个不封闭的弧形：

```
Arc2D.Double arc = new Arc2D.Double(125,125,60,50,90,135,Arc2D.OPEN);
```

这个弧形左上角的坐标是（125 ,125），宽 60 像素，高 50 像素。弧从 90 度开始，沿逆时针方向转过了 135 度。也可以通过使用 setArcByCenter 方法指定一个弧，例如：

```
    setArcByCenter(double x, double y, double radius, double angSt, double
angExt, int closure);
```

将此弧的位置、边界、角跨越和闭合类型设置为指定值。例如，可以重写面板对象的 paintComponent 方法（绘图代码必须放在 paintComponent 方法中），在该方法中，绘制一个弧形。

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;          //强制转换一个 Graphics2D 对象
    Arc2D.Double arc = new Arc2D.Double(125,125,60,50,90,135,Arc2D.OPEN);
    g2.draw(arc);
}
```

上面的代码中，创建了一个左上角在（125，125），宽 60 像素，高 50 像素，从 12 点



钟方向开始，逆时针转过 135 度的一个开放的弧形。

18.3.8　画二次曲线

二次曲线由实现了 Shape 接口的 QuadCurve2D 类代表。它有两个子类，QuardCurve2D.Float 和 QuardCurve2D.Double，分别代表不同精度类型的二次曲线。

创建二次曲线的语法格式如下。

```
QuadCurve2D.Double(double x1,double y1,                    //二次曲线的起始点
                   double ctrlx,double ctrly,              //中间控制点
                   double x2,double y2);                   //二次曲线的另一端点
QuadCurve2D.Float(float x1, float y1, float ctrlx, float ctrly,   float
x2, float y2);
```



图 18.6　二次曲线（红色的点为控制点）

其中前两个参数为二次曲线的起始端点，中间两个参数为曲线控制点，后两个参数为二次曲线的另个一端点，如图 18.6 所示。

例如可以使用如下的方式创建一个 double 精度的二次曲线。

```
QuadCurve2D q = new QuadCruve2D.Double(200, 200,
200,175,250,150);
```

可以使用 setCurve(QuadCurve2D c)，使用另一个二次曲线设置当前的二次曲线。例如：

```
QuadCurve2D q = new QuadCruve2D.Float();
//q.setCurve(x1,y1,ctrlx,ctrly,x2,y2); //方法中的参数对应两个端点和控制点坐标
q.setCurve(200, 200, 200,175,250,150);
g2.draw(q);;
```

可以重写面板对象的 paintComponent 方法（绘图代码必须放在 paintComponent 方法中），在该方法中，绘制一个二次曲线。例如：

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;          //强制转换一个 Graphics2D 对象
    //创建二次曲线对象
    QuadCurve2D q = new QuadCurve2D.Double(200, 200, 200,175,250,150);
    g2.draw(q);                             //画二次曲线
}
```

18.3.9　画三次曲线

三次曲线由实现了 Shape 接口的 CubicCurve2D 类代表。它有两个子类，CubicCurve2D.Float 和 CubicCurve2D.Double，分别代表不同精度类型的三次曲线。创建三次曲线的语法格式如下。

```
CubicCurve2D.Double(double x1,double y1,                   //三次曲线的起始点
                    double ctrlx1,double ctrly1,           //第一个中间控制点
                    double ctrlx2,double ctrly2,           //第二个中间控制点
                    double x2,double y2);                  //三次曲线的另一端点
CubicCurve2D.Float(float x1, float y1, float ctrlx, float ctrly,  float
x2, float y2);
```

其中前两个参数为三次曲线的起始端点，中间 4 个参数分别为两个曲线控制点，后两

个参数为三次曲线的另一端点，如图 18.7 所示。

例如可以使用如下的方式创建一个 double 精度的三次曲线：

```
CubicCurve2D q = new CubicCurve2D.Double(200, 200,
200,175,220,225,250,150);
```



图 18.7 三次曲线（红色的点为控制点）

可以使用 setCurve(QuadCurve2D c)，使用另一个二次曲线设置当前的二次曲线。例如：

```
CubicCurve2D q = new CubicCurve2D.Float();
//q.setCurve(x1,y1,ctrlx1,ctrly1,ctrlx2,ctrly2,x2,y2); //方法中的参数对应两
个端点坐标及两个中间控制点坐标
q.setCurve(200, 200, 200,175,220,225,250,150);
g2.draw(q);;
```

可以重写面板对象的 paintComponent 方法（绘图代码必须放在 paintComponent 方法中），在该方法中，绘制一个二次曲线。例如：

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;          //强制转换一个 Graphics2D 对象
    //创建三次曲线对象
    CubicCurve2D q = new CubicCurve2D.Double(200, 200, 200,175,220,225,250,
150);
    g2.draw(q);                             //画三次曲线
}
```

18.3.10 画任意图形

要画任意的图形，如不规则形状、多边形等，需要用到另一个类 GeneralPath，它实现了接口 Shape，代表根据直线、二次曲线和三次(Bézier)曲线构造的几何路径。它可以包含多个子路径。可以使用如下的方法构造一个空的 GeneralPath 实例：

```
GeneralPath gp = new GeneralPath();
```

在构造了 GeneralPath 实例实例对象 gp 后，可以使用下列方法向 GeneralPath 实例中添加路径片段。

- moveTo(float x,float y)：将路径的当前点移动到指定的坐标位置处。
- lineTo(float x,float y)：向当前路径中添加一个线段。
- quadTo(float ctrlx,float ctrly,float x2,float y2)：向当前路径中添加一个二次曲线段。
- curveTo(float ctrlx1,float ctrly1,float ctrlx2,float ctrly2,float x3,float y3)：向当前路径中添加一个三次曲线段。
- closePath()：封闭当前路径。

在构造 GeneralPath 对象时，根据需要可以指定其缠绕规则。缠绕规则指的是确定路径内部的方式。缠绕规则有两种类型：WIND_EVEN_ODD 和 WIND_NON_ZERO。

- WIND_EVEN_ODD 缠绕规则意味着，从路径外向区域内的点移动并穿过它时，路径的封闭区域在内部区域和外部区域之间交替变化。
- WIND_NON_ZERO 缠绕规则意味着，如果从给定点朝任意方向向无穷大绘制射线并检查路径与射线相交的位置，当且仅当路径从左到右穿过射线的次数不等于路径从右到左穿过射线的次数时，点位于路径内。



下面的代码演示了如何使用 GeneralPath 绘制一个不封闭的多边形。

```
    int x2Points[] = {0, 100, 0, 100};
    int y2Points[] = {0, 50, 50, 0};
    //创建路径对象，并确定路径内部的奇偶 (even-odd) 缠绕规则，指定初始的路径容量
    GeneralPath polyline = new GeneralPath(GeneralPath.WIND_EVEN_ODD, x2Points.
length);
    polyline.moveTo (x2Points[0], y2Points[0]);            //首先移动到原点
    for (int index = 1; index < x2Points.length; index++) {
         polyline.lineTo(x2Points[index], y2Points[index]);
                                                  //循环加入直线段路径
    };
    g2.draw(polyline);                            //绘制添加了直线段路径的路径对象
```

上面的代码绘制出的图形如图 18.8 所示。

下面的代码演示了如何使用 GeneralPath 绘制一个不封闭的多边形。

```
    int x1Points[] = {0, 100, 0, 100};
    int y1Points[] = {0, 50, 50, 0};
    GeneralPath polygon = new GeneralPath(GeneralPath.WIND_EVEN_ODD, x1Points.
length);
    polygon.moveTo (x1Points[0], y1Points[0]);
    for (int index = 1; index < x1Points.length; index++) {
         polygon.lineTo(x1Points[index], y1Points[index]);
    };
    polygon.closePath();                                  //封闭多边形
    g2.draw(polygon);
```

上面的代码绘制出的图形如图 18.9 所示。

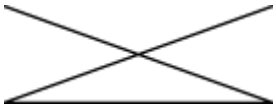


图 18.8　不封闭的多边形

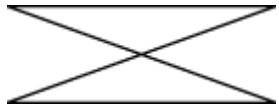


图 18.9　封闭的多边形

注意，这两行代码唯一的不同，在于封闭的多边形在添加完路径以后，需要使用 closePath()方法来封闭路径（在最后的线段的端点和上一次 moveTo()方法之间画一条直线进行封闭）。

可以使用 append()方法，将一个已经存在的几何形状追加到当前路径对象中，可能使用一条线段将新几何形状连接到现有的路径段。

18.3.11　设置画笔样式

除了绘制基本几何图形的轮廓之外，在 Java2D 中，还可以指定画笔的颜色、宽度和样式，也可以对几何图形的内部用指定填充模式进行填充。

要指定着色属性，使用 Graphics2D 类的 setColor()方法。例如，下面的代码，指定使用红色绘图。

```
    Graphics2D g2 = (Graphics2D)g;
    g.setColor(Color.RED);
```

在对几何图形进行渲染之前，先要对几何图形应用样式和填充模式，以改变

Graphics2D 上下文环境的画笔和绘图属性。例如，要画一个点虚线，要先创建一个相应的 Stroke 对象（代表设置了样式的画笔）。然后使用 Graphics2D 对象的 setStroke()方法将这个画笔添加到 Graphics2D 绘图环境中。在 Java2D 中，使用 BasicStroke 类代表一个画笔，设置和应用画笔按以下步骤：

① 在绘图之前，可以构造一个指定了相应属性（如线宽、线的终端样式以及两条线连接的样式）的画笔。

② 使用 Graphics2D 的 setStroke()方法，将构造好的画笔应用于 Graphics2D 绘图环境对象。

③ 创建要绘制的对象。可以是直线、矩形、椭圆、弧等几何图形。在上一节已经讲述了如何创建这些几何图形。

④ 绘制对象。所有的绘制对象使用相同的 Graphics2D 类的方法 draw()来画几何图形。

在构造画笔时，使用 BasicStroke 对象。一个 BasicStroke 对象持有线宽、线的连接样式、线的终端点的样式及点划线样式。当使用 draw()方法画图时，会使用这些信息。其中线宽是一个 float 类型的值，单位是像素。线的终端样式指的是线段的末尾的样式。BasicStroke 支持以下三种线的终端样式：

- BasicStroke.CAP_BUTT：BasicStroke 中定义的常量，代表没有终点。
- BasicStroke. CAP_ROUND：BasicStroke 中定义的常量，代表圆形终点。
- BasicStroke. CAP_SQUARE：BasicStroke 中定义的常量，代表方形终点。

三种线段终点样式如图 18.10～18.12 所示。



图 18.10 CAP_BUTT



图 18.11 CAP_ROUND



图 18.12 CAP_SQUARE

线的连接样式指的是两个线段相连时线段的末尾的样式。BasicStroke 支持以下三种连接样式：

- BasicStroke.JOIN_MITER：BasicStroke 中定义的常量，它通过扩展两条线段的外边界来将它们连接在一起，因此两条线连接处是尖角形式。
- BasicStroke.JOIN_ROUND：BasicStroke 中定义的常量，它将两条线段圆滑地连接起来。因此两条线连接处是圆角形式。
- BasicStroke.JOIN_BEVEL：BasicStroke 中定义的常量，它用一条直线将两条线段连接起来。因此两条线连接处是平角形式。

三种线段连接样式如图 18.13～18.15 所示。



图 18.13 JOIN_MITER



图 18.14 JOIN_ROUND



图 18.15 JOIN_BEVEL

点划线样式定义一条线段透明和不透明间隔的模式，如虚线。点划线样式定义用到一个点划线数组和点划线开始的偏移量。其中点划线数组定义虚线模式，数组中元素交替代表虚线的长度和虚线之间空白的长度（以像素为单位）。例如，数组的元素 0 代表第一个点，元素 1 代表第一个空白，依此类推。而偏移量代表虚线模式中哪一部分应用于线的开始。定义一个画笔 BasicStroke 对象的语法如下所示。

```
public BasicStroke(float width,
                   int cap,
                   int join,
                   float miterlimit,
                   float[] dash,
                   float dash_phase);
```

这个构造器构造一个具有指定属性的新 BasicStroke。其中各参数的含义如下。

- width：此 BasicStroke 的宽度。此宽度必须大于或等于 0.0f。如果将宽度设置为 0.0f，则将笔划呈现为可用于目标设备和抗锯齿提示设置的最细线条。
- cap：BasicStroke 端点的装饰。
- join：应用在路径线段交汇处的装饰。
- miterlimit：斜接处的剪裁限制。miterlimit 必须大于或等于 1.0f。默认为 10.0f。
- dash：表示虚线模式的数组。
- dash_phase：开始虚线模式的偏移量。

上面的各参数中，最后的两个参数也可以省略，这时创建的是实线。例如，下面的 paintComponent()方法中，创建一个画笔 BasicStroke 对象，然后使用此画笔绘制一个圆角矩形。

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;              //强制转换 Graphics2D 对象
    final static float dash1[ ] = {10.0f};      //指定虚线模式的数组
    final static BasicStroke dashed = new BasicStroke(1.0f,    //指定线宽
                                  BasicStroke.CAP_BUTT, //指定线的终点样式
                                  BasicStroke.JOIN_MITER,
                                                //指定两条线连接时尾部的样式
                                  10.0f, dash1, 0.0f);//指定虚线模式和偏移量
    g2.setStroke(dashed);                       //将构建的画笔应用于绘图环境
    g2.draw(new RoundRectangle2D.Double(x, y,50,20,10,10));
                                                //绘制一个圆角矩形
}
```

绘制的图形如图 18.16 所示。



图 18.16　虚线圆角矩形

18.3.12　设置填充模式

填充指的是以一种固定的颜色或一种渐变的颜色，或一个纹理模式，来绘制图形的内部。填充模式控制着一个绘图对象如何被填充，包括填充颜色、渐变填充、纹理模式等。

在 Java2D 中，使用实现了 Paint 接口的类的实例代表一个填充模式。在 Java2D 中，有三个实现了 Paint 接口的类：Color，GradientPaint 和 TexturePaint。其中 GradientPaint 代表渐变填充，TexturePaint 代表纹理模式。

渐变填充：指的是从一个坐标点上的某种颜色逐渐变化到另一个坐标点上的另一种颜色。这种转变可以在两个点之间发生一次，这叫做一个非周期性渐变。或者也可以发生多次，这叫做一个周期性渐变。

按以下步骤设置和应用填充模式：

① 在绘图之前，可以构造一个模式对象。

② 使用 Graphics2D 的 setPaint()方法，将构造好的填充模式应用于 Graphics2D 绘图环境对象。

③ 创建要填充的图形对象。可以是矩形、椭圆、封闭弧等几何图形。在上一节已经讲述了如何创建这些几何图形。

④ 绘制对象。所有的绘制对象使用相同的 Graphics2D 类的方法 fill()来填充几何图形。

定义一个 GradientPaint 填充对象的语法如下所示。

```
public GradientPaint (float x1,float y1,    //渐变颜色开始的坐标位置
                int color1,                 //开始的颜色
                float x2,float y2,          //渐变颜色结束的坐标位置
                int color2,                 //结束的颜色
                boolean isCyclic);          //可选。如果为 true，则为周期性渐变
```

在默认情况下，这个构造器构造一个非周期性的渐变填充对象。如果在最后加上一个 true 参数，则构造一个周期性的渐变填充对象。定义一个 TexturePaint 纹理填充对象的语法如下所示。

```
public TexturePaint ((BufferedImage txtr,  //作为纹理的图像
                Rectangle2D ancho);        //用来重复和定位纹理图像的矩形区域
```

计算纹理在设备空间中的位置的方式，是在用户空间中的所有方向上无限制地复制指定 Rectangle2D，然后将 BufferedImage 映射到各个复制的 Rectangle2D。

例如，下面的 paintComponent()方法中，创建一个渐变填充模式和一个椭圆 Ellipse2D.Double 对象，然后使用此填充模式填充这个椭圆图形。

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;              //强制转换 Graphics2D 对象
    redtowhite = new GradientPaint(0,0,Color.RED,100, 0,Color.WHITE);
                                                //构造一个渐变填充对象
    g2.setPaint(redtowhite);                            //设置填充模式
    g2.fill (new Ellipse2D.Double(0, 0, 100, 50));      //填充椭圆
}
```

绘制的图形如图 18.17 所示。

18.3.13 实例：绘制和填充几何图形

Java 2D API 在构建绘图程序时特别有用。本节通过一个综合示例 DrawGeometryDemo.java，演示绘制和填充各种几何图形的方法。

【例 18.2】编写一个应用程序的示例。程序的功能是在面板上绘制各种基本的几何图形和多边形，如图 18.18 所示。



图 18.17 渐变填充

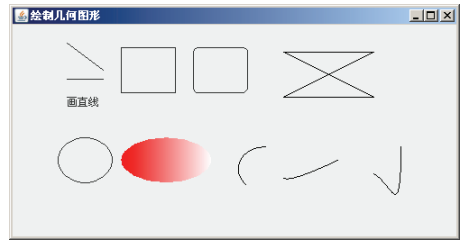


图 18.18 画几何图形和多边形



实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter18，并在 chapter18 目录下，创建 java 源文件 DrawGeometryDemo.java；

② 使用记事本，打开 SimpleDrawDemo.java 源文件进行编辑，输入以下代码：

```
package chapter18;      //创建 chapter18 包
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
```



程序放在包 chapter18 中；需要导入 java.awt 包、javax.swing 包和 import java.awt.geom 包。

③ 接着编写自定义的 JPanel 类，并重写其 paintComponent()方法，将绘制和填充几何图形的方法放在这个方法中。代码如下。

```
//自定义面板,用来作图
class MyPane extends JPanel{
   public void paintComponent(Graphics g){
        Graphics2D g2 = (Graphics2D)g;
        //画直线
        Line2D.Double line1 = new Line2D.Double(60,20,100,50);
        Line2D.Double line2 = new Line2D.Double();
        line2. setLine(60,60,100,60);
        g2.draw(line1);
        g2.draw(line2);
        g2.drawString("画直线",60,90);
        //画直角矩形
        Rectangle2D.Double rect = new Rectangle2D.Double(120,25,60,50);

        g2.draw(rect);                                  //画直角矩形
        //画圆角矩形
        RoundRectangle2D.Double roundrect = new RoundRectangle2D.Double
(200,25,60,50,10,10);
        g2.draw(roundrect);                             //画圆角矩形
        //画椭圆形
        Ellipse2D.Double ellipse = new Ellipse2D.Double(50,125,60,50);
        g2.draw(ellipse);                               //画椭圆形
        //填充椭圆
        GradientPaint redtowhite = new GradientPaint(120,150,
                                                        //构造一个渐变填充对象
                                    Color.RED,
                                    220, 150,
                                    Color.WHITE);
        g2.setPaint(redtowhite);                        //设置填充模式
        g2.fill (new Ellipse2D.Double(120, 125, 100, 50));     //填充椭圆
        //恢复画笔为黑色
        g2.setPaint(Color.black);
        //画弧形
        Arc2D.Double arc = new Arc2D.Double(250,135,60,50,90,135,Arc2D. OPEN);
        g2.draw(arc);
        //画二次曲线
```

```
            QuadCurve2D qc = new QuadCurve2D.Double(300, 170, 310,175,360,150);
            g2.draw(qc);                                    //画二次曲线
            //画三次曲线
            CubicCurve2D cc = new CubicCurve2D.Double(400, 165, 420,175,430,
225,430,135);
            g2.draw(cc);                                    //画三次曲线
            //画多边形
            int x1Points[] = {300, 400, 300, 400};
            int y1Points[] = {30, 80, 80, 30};
            GeneralPath polygon = new GeneralPath(GeneralPath.WIND_EVEN_ODD,
x1Points.length);
            polygon.moveTo (x1Points[0], y1Points[0]);
            for (int index = 1; index < x1Points.length; index++) {
                polygon.lineTo(x1Points[index], y1Points[index]);
            }
            polygon.closePath();                            //封闭多边形
            g2.draw(polygon);                               //画多边形
        }
    }
```

同样，在自定义面板时，需要继承自 swing 的 JPanel 类，并且重写 paintComponent() 方法，在此方法中获得绘图上下文环境 Graphics，并强制转换为 Graphics2D 对象，然后使用 Graphics2D 的绘图功能绘制各种图像。

④ 编写程序窗体界面类 DrawGeometryFrame，该类继承自 JFrame 类。在该类中添加了一个自定义面板对象 myPane。具体代码如下。

```
//编写应用程序 DrawGeometryFrame
class DrawGeometryFrame extends JFrame{
    DrawGeometryFrame(){
        super();
        setTitle("绘制和填充几何图形");
        setBounds(100,100,500,300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//设置默认窗口关闭操作
        MyPane myPane = new MyPane();                   //创建自定义面板对象
        this.getContentPane().add(myPane,BorderLayout.CENTER);
                                                        //添加自定义组件
        this.setVisible(true);                          //显示窗体
    }
}
```

程序窗体界面类 DrawGeometryFrame 的构造器中完成组件对象初始化，并完成界面布局。

⑤ 创建主程序 DrawGeometryDemo，具体代码如下。

```
//编写主程序
public class DrawGeometryDemo{
    public static void main(String[] args){
        new DrawGeometryFrame();                //创建主程序类对象，运行程序
    }
}
```

⑥ 使用下面的命令编译并运行主程序 DrawGeometryDemo.java。

```
javac -d . DrawGeometryDemo.java
java chapter18. DrawGeometryDemo
```



编译并运行以后，就会看到如图 18.17 所示的程序运行界面。可以看到，在容器面板中已经有了绘制好的各种几何图形。



18.4　绘制文本

在前几节的示例中，已经使用了绘制文本的方法 drawString()。在 Java 中，还可以指定要绘制的文本的字体，获取本地系统已安装的字体等。本节介绍如何获取和选择字体，如何设置字体，以及如何绘制文本内容。

18.4.1　选择字体

在 Java 中，使用 java.awt.Font 类对象来代表字体。Font 类对象表示一个字体的名字、样式和大小。Java2D 中定义了以下 5 种逻辑字体：

- Dialog；
- DialogInput；
- Monospaced；
- Serif；
- SansSerif。

这些字体在任何 Java 平台上都是可用的。例如，Serif 字体与 Times New Roman 相似，通常用于打印输出。而一个 Sans Serif 字体更多地应用于屏幕字体。

除了字体名字之外，Font 类还有另外两个很重要的属性：样式和大小。Font 有三种样式，可以通过使用常量 Font.PLAIN、Font.BOLD 和 Font.ITALIC 来设定。这些值都是整数值，分别代表普通文本、黑体和斜体。也可以将它们叠加在一起来得到组合的效果。默认的 Java 2D 的字体大小是 12pt Dialog。在应用程序中可以直接以如下的方式创建一个这种字体的实例对象：

```
Font font1 = new Font("Dialog",Font.PLAIN,12);
```

下面的代码创建一个 24pt 的 Dialog 字体，同时将其样式设为既是黑体又是斜体。

```
Font font2 = new Font("Dialog",Font.BOLD+Font.ITALIC,24);
```

创建字体对象以后，可以使用 Graphics2D 类的 setFont()方法，并将创建的 Font 对象作为其参数，设置当前使用的字体。下面的代码使用前面创建的字体 font2 来设置字体。

```
g2.setFont(font2);
```

接着调用 Graphics2D 的 drawString()方法绘制字符串。这个方法使用当前选定的字体。如果没有选定字体，会使用默认的字体。一个新的字体可以使用 setFont()方法在任何时候设定。例如，可以重写面板对象的 paintComponent()方法，在该方法中，绘制一个字符串。

```
public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D)g;          //强制转换一个 Graphics2D 对象
    //g2.setColor(Color.RED);               //可以使用 setColor 方法设置颜色
    g2.setPaint(Color.blue);                //也可以使用 setPaint 方法设置颜色
    Font font = new Font("Dialog",Font.BOLD+Font.ITALIC,24); //创建一个字体
    g2.setFont(font);                       //设置当前字体
    g2.drawString("大家好！",50,100);        //绘制字符串
```

```
}
```

在绘图之前，可以设置要使用的颜色，这需要设置绘图环境 Graphics2D 的颜色属性。既可以使用其 setColor 方法，也可以使用 setPaint()方法，因为 Color 对象也是实现了 Paint 接口的对象。



物理字体是实际的字体库，包含字形数据和表，这些数据和表使用字体技术将字符序列映射到字形序列。通常，每种物理字体只支持有限的书写系统集合，例如，只支持拉丁文字符，或者只支持日文和基本拉丁文。可用的物理字体集合随配置的不同而有所不同。逻辑字体是由必须受所有 Java 运行时环境支持的 Java 平台所定义的 5 种字体系列：Serif、SansSerif、Monospaced、Dialog 和 DialogInput。这些逻辑字体不是实际的字体库。此外，由 Java 运行时环境将逻辑字体名称映射到物理字体。映射关系与实现和通常语言环境相关，因此它们提供的外观和规格各不相同。

18.4.2 获得本地系统已安装的字体

除了 Java 的逻辑字体之外，还可以访问安装到本地系统中的其他字体。这需要用到类 java.awt.GraphicsEnvironment。GraphicsEnvironment 类描述了 Java 应用程序在特定平台上可用的 GraphicsDevice 对象和 Font 对象的集合。此 GraphicsEnvironment 中的资源可以是本地资源，也可以位于远程机器上。



GraphicsDevice 类描述可以在特定图形环境中使用的图形设备。GraphicsDevice 对象可以是屏幕、打印机或图像缓冲区，并且都是 Graphics2D 绘图方法的目标。这些设备包括屏幕和打印机设备。在 GraphicsEnvironment 实例中可以有多个屏幕和多个打印机。

GraphicsEnvironment 是一个抽象类，不能直接创建其实例，可以通过如下的方式获得一个 GraphicsEnvironment 对象：

```
GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
```

静态方法 getLocalGraphicsEnvironment()返回本地图形设备环境，它封装了本地图形设备环境的信息。调用其 getAvailableFontFamilyNames()方法可以获得本地系统安装的所有字体名称。例如：

```
String[ ] fontFamilies = ge.getAvailableFontFamilyNames();
```

该方法返回一个字体名称的数组。

【例 18.3】编写一个应用程序的示例。程序的功能是获取本地系统安装的所有字体并在控制台输出显示。

```
import java.awt.*;
public class LocalAvailableFontFamily{
    public static void main(String[ ] args){
        GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphics
Environment();
        String[ ] fontFamilies = ge.getAvailableFontFamilyNames(); //获取
```

```
本地系统可用的字体
            for(int i=0;i<fontFamilies.length;i++)
                System.out.println(fontFamilies[i]);    //循环输出每一种字体
        }
    }
```

在这个程序中，获得本地系统安装的所有字体的一个数组，然后遍历此数组，依次输出其中的每一种字体的名称。

18.4.3　创建一个新的字体

有时候，不想使用系统已经安装的字体，例如想使用字体库导入的另外的自定义的字体，这时可以在应用程序中自己创建一种新字体。Java 平台的所有实现都支持 TrueType 字体，这是在现代操作系统中最普遍使用的字体。对于这种要求特定字体的应用程序可以使用 Font 类的静态方法 createFont()来捆绑这些字体，并对其进行实例化。

createFont 方法的定义如下。

```
//返回一个使用指定字体类型和指定字体文件的新 Font。
static Font createFont(int fontFormat, File fontFile);
//返回一个使用指定字体类型和输入数据的新 Font。
static Font createFont(int fontFormat, InputStream fontStream);
```

要创建一个 TrueType 字体，fontFormat 参数必须是常量 Font.TRUETYPE_FONT。例如：

```
Font font = Font.createFont(Font.TRUETYPE_FONT,new File("A.ttf");
```

直接从文件中访问字体更加方便，但是，如果不能访问文件系统资源，或者字体库是被打包进一个 Java 档案文件（JAR）中时，就需要使用 InputStream（关于流，会在第 20 章介绍）。

使用 createFont()方法返回的字体并不会自动地被字体系统所识别，因为由应用程序所创建的字体并不是固有字体的一部分，所以要使用 Font 类的 deriveFont()方法很重要。使用 deriveFont()方法可以设置新的字体的大小和样式。例如下面的代码将新创建的字体大小设置为 12：

```
try {
    Font font = Font.createFont(Font.TRUETYPE_FONT, new File("A.ttf"));
//默认字体大小为 1pt
    return font.deriveFont(12f);                //将字体大小设为 12pt
}catch(Exception e){};
```

创建完自定义的字体之后，需要向当前的图形设备环境注册此字体。例如：

```
try {
    GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphics
Environment();
    ge.registerFont(Font.createFont(Font.TRUETYPE_FONT, new File("A.ttf"));
                                                        //注册字体
}catch(Exception e){};
```

上面的 try…catch 语句为异常捕获语句。在创建自定义的字体时，有可能因为某些原因创建失败，例如找不到 A.ttf 文件。异常处理会在下一章介绍。

在注册后，再使用 GraphicsEnvironment 的 getAvailableFontFamilyNames()方法就可以

获得了，并且也可以被用于 Font 的构造器中。

18.4.4 获取某个字体的信息

使用类 java.awt.FontMetrics 可以得到当前字体的具体信息，如字体的高度或字符的宽度。要使用这个类的方法，必须使用 Graphics 类的方法 getFontMetrics()来创建一个 FontMetrics 的对象。这个方法需要一个字体 Font 对象作为参数。如下面的代码所示。

```
FontMetrics fm = graphics.getFontMetrics(font);
                          //从 Graphics 类获得 FontMetrics 对象,它代表字体的规格
int hgt = fm.getHeight();                    // 获得字体的高度
int adv = fm.stringWidth(text);              //返回一个字符串 text 的全部宽度
int wet = fm.charWidth(charact);             //返回一个给定字符的宽度
```

在上面的方法中，获得的字符的高度和宽度均以像素为单位。

 18.5 绘制图像

在 Java 中，基本的图像处理是通过类 java.awt.Image 和 java.awt.image.BufferedImage 来完成的。Image 类代表的图像有一定的大小（长和宽），以像素为单位，并有独立于绘图表面的坐标系统。Image 对象可以表示三种格式的图像：GIF，JPEG 和 PNG。BufferedImage 类继承自 Image 类，它允许应用程序直接操作图像数据（例如，获取或设置像素的颜色）。

18.5.1 读取/加载图像

要想在程序中使用 JPEG 格式（通常用于数码相片的格式）和 GIF 格式（通常用于 web 页面的图像格式）的图像，首先必须将它们从外部格式转换为内部格式。

Java 2D 支持使用其 Image I/O API 将这些外部图像加载到其 BufferedImage 格式中。Image I/O API 位于 javax.imageio 包中，它支持 GIF、PNG、JPEG、BMP 和 WBMP 格式。

使用类 ImageIO 的静态成员方法 public static BufferedImage read()可以从图像文件中读取图像数据到内存即类 java.awt.image.BufferedImage 的实例对象中。例如，使用如下的代码从一个指定的文件中加载一个图像：

```
BufferedImage img = null;
try {
    img = ImageIO.read(new File("star.jpg"));   //读取 star.jpg 图像文件
} catch (IOException e) {}                      //捕获读取文件是发生的 IO 异常
```

Image I/O 能够识别出 JPEG 格式图像文件的内容，并解码到一个可以被直接使用的 BufferedImage 对象中。

如果是在小应用程序中加载图像，使用如下的代码：

```
try {
    URL url = new URL(getCodeBase(),"star.jpg");
    img = ImageIO.read(url);                    //读取 star.jpg 图像文件
} catch (IOException e) {}                      //捕获读取文件是发生的 IO 异常
```

除了从文件和 URL 中读取图像文件外，Image I/O 也可以从 InputStream 中读取。另外，应用程序可以直接构造 BufferedImage 类的实例。BufferedImage 可以在内存中管理图像并提供



相应的方法保存、解释和获取像素数据。因为 BufferedImage 是 Image 的子类，所以凡能接受 Image 参数的 Graphics 和 Graphics2D 的方法，都可以接受 BufferedImage 参数。BufferedImage 本质上是带有一个可访问数据缓冲区的 Image，因此直接使用 BufferedImage 类会更有效率。

18.5.2　绘制图像

加载了图像到程序中以后，可以使用 Graphics 的 drawImage()方法在指定的位置绘制此图像。drawImage()方法的定义如下所示。

```
public abstract boolean drawImage(Image img,
                                  int x,
                                  int y,
                                  ImageObserver observer);
```

参数说明如下。

- img：代表当前要绘制的图像。如果 img 为 null，则此方法不执行任何操作。
- x,y：代表图像的左上角在图形上下文坐标空间的坐标。
- observer：加载和转换了更多图像时要通知的对象。

如果图像已经完整加载，并且其像素不再发生更改，则 drawImage()返回 true。否则，drawImage()返回 false，并且随着更多的图像可用或者到了绘制动画另一帧的时候，加载图像的进程将通知指定的图像观察者。observer 参数经常并不直接使用，并且对于 BufferedImage 类来说不需要，所以这个参数通常为 null。

上面的 drawImage()方法默认情况下按 1:1 绘制图像。有时需要绘制图像的一部分（子图像），或者缩放图像，或者使图像变形或过滤图像，这需要使用另一个重载的 drawImage()方法。

```
public abstract boolean drawImage(Image img,           //要绘制的图像
                                  int dstx1, int dsty1, int dstx2, int dsty2,
                                                       //绘制的目标区域
                                  int srcx1, int srcy1, int srcx2, int srcy2,
                                                       //拷贝的源图像区域
                                  ImageObserver observer);
```

参数说明：

- img：要绘制的指定图像。如果 img 为 null，则此方法不执行任何操作。
- dstx1：目标矩形第一个角的 x 坐标。
- dsty1：目标矩形第一个角的 y 坐标。
- dstx2：目标矩形第二个角的 x 坐标。
- dsty2：目标矩形第二个角的 y 坐标。
- srcx1：源矩形第一个角的 x 坐标。
- srcy1：源矩形第一个角的 y 坐标。
- srcx2：源矩形第二个角的 x 坐标。
- srcy2：源矩形第二个角的 y 坐标。
- observer：当缩放并转换了更多图像时要通知的对象。

此方法执行图像从源到目标的缩放：源矩形的第一个坐标被映射到目标矩形的第一个坐标，第二个源坐标被映射到第二个目标坐标。按需要缩放和翻转子图像以保持这些映射关系。

18.5.3 创建并绘制图像

除了从外部加载图像之外，也可以在内存中创建一个全新的图像。这种手工创建图像的步骤如下。

① 使用 new BufferedImage(int width,int height,int type)构造器在内存中构造一个 BufferedImage 对象，其中前两个参数指定图像的宽和高，第三个参数指定图像的类型，它是一个常量，指的是颜色的表示方法。此参数的常用值有 BufferedImage.TYPE_INT_RGB 和 BufferedImage.TYPE_INT_ARGB。

② 使用 BufferedImage 的 createGraphics()方法，获得此图像的一个 Graphics2D 对象，可以将内存中的图像本身看做一个绘图表面。

③ 最后，使用获得的此图像的 Graphics2D 对象在内存中绘制此图像。

例如，在下面的代码中，在内存中创建了一个图像：

```
...
BufferedImage off_Image =
    new BufferedImage(100, 50, BufferedImage.TYPE_INT_ARGB);
                                                    //在内存中创建一个图像
Graphics2D g2 = off_Image.createGraphics();         //获得此图像的绘图上下文
```

在内存中创建了一个 RGB 类型的图像，并使用 createGraphics()方法获得此图像的绘图上下文环境。随后可以使用此上下文环境在内存中的当前图像上直接绘图。

18.5.4 写入/保存图像

不管是通过 Image I/O 类从外部的图像文件加载到内存的原始图像，还是在内存中已经修改过的图像或在内存中直接创建的图像，它们都是 BufferedImage 对象，都可以保存到外部图像文件中。

ImageIO 类有一个静态的方法 write()，可以很方便地将 BufferedImage 对象以不同的图像格式保存，其语法格式如下。

```
public static boolean write(RenderedImage im,String formatName,File output)
throws IoException;
```

该方法中的第一个参数是要绘制的 BufferedImage 对象，它实现了 RenderedImage 接口。第二个参数 formatName 代表要保存的图像格式。例如下面的代码将一个 BufferedImage 对象保存为 png 格式的文件。

```
try {
    BufferedImage image = ImageIO.read(new File("star.jpg"));
                                  //读取一个 jpg 格式的文件到 Bufferedimage 中
    File outputfile = new File("saved.png");   //创建一个文件对象 saved.png
    ImageIO.write(image, "png", outputfile);
                                  //将 image 对象以 png 格式保存到 saved.png 中
} catch (IOException e){ }        //文件读取过程有可以发生异常，所以要捕获
```

ImageIO 类支持读取多种格式的图像文件。



18.5.5　获取可读写的图像类型

使用类 ImageIO 的静态成员方法 public static String[] getReaderFormatNames()，以字符串数组的形式返回类 javax.imageio.ImageIO 所支持的所有图像输入格式。例如：

```
String[] readerFormatNames = ImageIO.getReaderFormatNames();
```

类 ImageIO 的静态成员方法 public static String[] getWriterFormatNames()，以字符串数组的形式返回类 javax.imageio.ImageIO 所支持的所有图像输出格式。

```
String[] writerFormatNames = ImageIO.getWriterFormatNames();
```

使用 getWriterFormatNames()方法返回的名称数组包含所有可支持的图像格式，这些格式名称可用于 writer 方法中，指定要输出的图像格式。

【例 18.4】编写一个简单的应用程序示例。程序的功能是输出当前支持的可读写的图像类型：

```
package chapter18;
public class ImageFormatNames{
    public static void main(String[] args){
        //获取可读取的图像类型
        String[] readerFormatNames = ImageIO.getReaderFormatNames();
        System.out.print("当前支持的可读取的图像类型有：");
        //循环输出每一种可读取的图像类型
        for(int i=0;i<readerFormatNames.length;i++){
            System.out.print(readerFormatNames[i] + " ");
        }
        System.out.println();                                    //换行
        //获取可写入的图像类型
        String[] writerFormatNames = ImageIO.getWriterFormatNames();
        System.out.print("当前支持的可写入的图像类型有：");
        //循环输出每一种可读取的图像类型
        for(int i=0;i<writerFormatNames.length;i++){
            System.out.print(writerFormatNames[i] + " ");
        }
    }
}
```

在上面的程序中，使用 ImageIO 类来获取当前系统支持的可读取和可写入的图像类型，并使用 for 语句在控制台循环输出每一种可读取的图像类型。编译并运行此程序，输出结果如下所示。

```
当前支持的可读取的图像类型有：jpg BMP bmp JPG wbmp jpeg png PNG JPEG WBMP GIF gif
当前支持的可写入的图像类型有：BMP bmp jpg JPG wbmp jpeg png PNG JPEG WBMP GIF gif
```

18.5.6　实例：读写 jpg 图像

在开发过程中，为了使程序界面更加美观，更加吸引人，经常需要在程序界面中点缀一些图像。本节就通过一个综合示例 LoadImageDemo.java，演示使用 ImageIO 对图像进行读取和绘制的方法。

【例 18.5】编写一个简单的应用程序示例。程序的功能是读取并绘制图像，如图 18.19

所示。

实现此程序的步骤如下。



图 18.19　绘制图像

① 在 D 盘根目录下，创建文件夹 chapter18，并在 chapter18 目录下，创建 java 源文件 LoadImageDemo.java；

② 使用记事本，打开 LoadImageDemo.java 源文件进行编辑，输入以下代码：

```
package chapter18;      //创建 chapter18 包
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
import javax.swing.*;
```



程序放在包 chapter18 中；需要导入相关的包，特别注意，ImageIO 类位于 javax.imageio 包中。

③ 接着编写自定义的 LoadImageDemo 类，它是 Component 类的子类，在其构造器中使用 ImageIO 类加载 images 文件夹下的图像文件 strawberry.jpg，并保存到 BufferedImage 对象中。需要注意的是，images 文件夹必须与 LoadImageDemo.java 源文件位于同一目录下。如果文件路径不对，或者文件找不到，那么将会产生一个 IO 异常，需要在代码中捕获。其代码如下。

```
//自定义一个 LoadImageDemo 类,重写其 paint 方法
public class LoadImageDemo extends Component {
    BufferedImage img;
    public LoadImageDemo() {                    //加载图像
      try {
          //读取 images 文件夹下的 strawberry.jpg 文件
          img = ImageIO.read(new File("images/strawberry.jpg"));
      } catch (IOException e) { }
    }
    ……
}
```

④ 接着重写 Component 组件的 paint()和 getPreferredSize()方法，重定义组件 LoadImageDemo 的大小和绘制方式。将绘制图像的方法放在 paint()方法中。代码如下。

```
//自定义一个 LoadImageDemo 类,重写其 paint 方法
public class LoadImageDemo extends Component {
    BufferedImage img;
    public LoadImageDemo() {                    //加载图像
      try {
          //读取 images 文件夹下的 strawberry.jpg 文件
          img = ImageIO.read(new File("images/strawberry.jpg"));
      } catch (IOException e) { }
    }
    //重写此组件的 paint 方法
    public void paint(Graphics g) {
       g.drawImage(img, 25, 0, null);           //绘制指定图像
    }
```



```
    public Dimension getPreferredSize() {          //覆盖 Component 类的同名方法
       if (img == null) {                          //如果图像读取不成功
            return new Dimension(100,100);         //返回大小为 100*100
       } else {                                    //如果图像读取成功
          //返回实际图像的大小
          return new Dimension(img.getWidth(null), img.getHeight(null));
       }
    }
    ……
}
```

在上面定义的 LoadImageDemo 类继承自欺欺人 Component 类，所以要重写其 paint()方法。而对 getPreferredSize()方法的重写，返回一个表示当前组件首选大小的 Dimension 对象。

⑤ 编写 main()方法，创建程序主窗体并添加自定义的组件 LoadImageDemo。其代码如下。

```
//编写 main 方法，创建窗体并添加组件
public static void main(String[] args) {          //程序主方法
       JFrame f = new JFrame("加载图像示例");
       f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       f.add(new LoadImageDemo());                 //添加组件
       f.pack();                                   //设置组件以首选大小显示
       f.setVisible(true);                         //显示窗体
}
```

⑥ 使用下面的命令编译并运行主程序 LoadImageDemo.java。

```
javac -d . LoadImageDemo.java
java chapter18. LoadImageDemo
```

编译并运行以后，就会看到如图 18.19 所示的程序运行界面。一定要注意，在上面的程序中，读取一个 images 文件夹下的 JPG 图片，而 images 文件夹一定要与 LoadImageDemo.java 源文件在同一目录下。否则，程序将会出现运行时异常，报告找不到输入文件。



18.6　小结

本章介绍了 Java 中对图形、文本和图像的处理，主要是使用 Java 2D API。在本章中，对图像的处理就是读者学习的重点，因为在实际开发中，相对图形和文本的处理，可能使用 Java 的机会并不是很多。其中文本的处理较为复杂，是读者学习的难点。建议读者能理解本章的内容即可。下一章将介绍 Java 程序中的异常处理。

4

第 4 篇

Java 开发技术篇

- 第 19 章　异常处理
- 第 20 章　输入与输出
- 第 21 章　并发程序设计
- 第 22 章　网络编程
- 第 23 章　Java 数据库应用程序开发基础
- 第 24 章　使用 Swing 组件创建数据库应用程序

第19章 异常处理

天有不测风云，人有旦夕祸福，Java 的程序代码也是如此。虽然每个程序员都想编写出完美而没有 bug 的代码，但实际上，这似乎是一种幻想。对于程序员来说，在编程过程中，首先应当尽可能去避免错误和异常的发生，对于不可避免、不可预测的情况应在异常发生时考虑如何处理。



19.1 Java 异常

异常是程序中的一些错误，但并不是所有的错误都是异常，并且错误有时候是可以避免的。有些异常需要处理，有些则不需要处理，为了加强程序的键壮性，程序设计时必须考虑到可能发生的异常事件并做出相应的处理。在 Java 语言中，产生异常现象的错误大体分为三类：编译错误、运行错误和逻辑错误。

19.1.1 Java 程序的编译错误

编译错误是由于所编写的程序存在语法问题，未能通过由源代码到字节码的编译而产生的，它由语言的编译系统负责检测和报告。此类错误在编译的时候会被检查出来，并不会生成运行代码，只有更正程序中的语法问题后才可以运行程序。下面列举部分常见编译错误。

1．大小写混淆

Java 语言是严格区分大小写的计算机编程语言。类、方法、变量的名称必须前后完全一致，否则将出现无法解析符号的错误。例如，声明了变量 mypet，在使用变量时书写为 myPet，则编译时会显示如图 19.1 所示的错误。此外，语句命令同样要注意大小写，否则会出现命令不存在的错误，如图 19.2 所示。

2．数据类型与变量类型不符

当数据类型与变量类型不符时，如果 Java 语言可以自行转换的话，则不会显示错误提示信息，但是运行结果将很可能与编程者的意愿不符。如果 Java 语言不可以自行转换，则会显示错误。例如，下面语句的错误信息如图 19.3 所示。

```
int number=55.5;//显示可能丢失精确度的错误信息
```

```
String str='A';//显示类型不兼容的错误提示
```



图 19.1　大小写混淆（1）



图 19.2　大小写混淆（2）

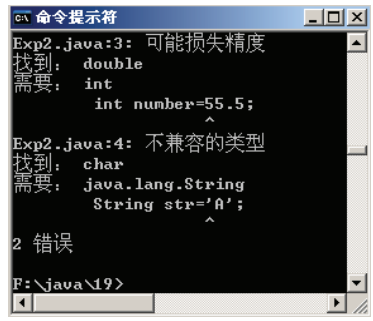


图 19.3　数据类型与变量类型不符

3．使用未声明的变量

在有些高级语言中，变量不需要先声明就可以直接使用。而 Java 语言规定在使用任何变量前必须要先声明该变量的类型。如果使用未声明的变量，编译程序时会出现无法解析符号的错误。

19.1.2　Java 程序的运行错误

运行错误是指程序在执行过程中发生的错误，它会中断程序的正常执行。例如，计算时除数为零、数组下标越界、文件没找到等。这类错误在编译程序时一般是无法发现的。下面列出两种常见的运行错误。

1．数组下标越界

数组下标越界是初学者常犯的一个错误。Java 语言规定数组的下标从 0 开始到数组元素个数减 1 为止。例如，下面程序中数组 numbers 共有 10 个元素，给这 10 个元素赋值的循环语句如下。

```
class Exp3{
    public static void main(String [] args){
        int[ ] numbers = new int[10];          //声明数组 numbers，大小为 10
        for(int i=1;i<=10;i++)                 //从 1 开始循环
            numbers[i] = 100;
    }
}
```

实际上，数组 numbers 的下标范围是 0~9，所以当执行最后一次循环 i=10 时，numbers[10] 已经超出了数组下标范围，会显示数组下标溢出范围的错误信息，如图 19.4 所示。

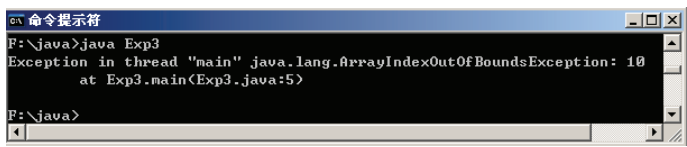


图 19.4　数组下标越界

2．除数为零

Java 语言中不允许计算过程中出现除数为零的情况，因此，在编写程序时一定要注意

计算过程中的中间值。如果出现除数为零的情况，则程序会被中断并显示除数为零的错误信息。例如，下面的程序段中给变量 i 赋初值为零，然后进行除运算。

```
class Exp4{
   public static void main(String [] args) {
         int i=0;
         i=3/i;
   }
}
```

运行时得到如图 19.5 所示的错误信息。

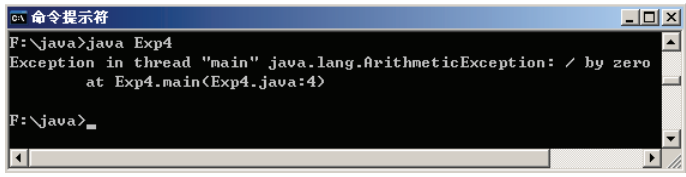


图 19.5　除数为零

运行错误通常都比较隐蔽，而且会造成程序中断甚至系统死机等现象。为此 Java 语言提供处理这类错误的方法，帮助编程者避免程序中断等现象的发生。

19.1.3　Java 程序中的逻辑错误

逻辑运行错误是指程序不能实现编程人员的设计意图和设计功能而产生的错误，即程序运行后没有得到预期的结果。该类错误从语法上来说是有效的，只是程序在逻辑上存在着缺陷。通常，逻辑错误不会产生错误提示信息，所以错误较难排除。下面列出两种常见的逻辑错误。

1．超出数据类型的取值范围

在 Java 中，每种数据类型都有其取值范围，一旦数值超出了数据类型的取值范围，就会造成计算结果的错误。例如，下面的循环语句用来求 20!。

```
int sum=1;
for(int i=1;i<=20;i++)
  sum=sum*i;
```

变量 sum 的计算结果为-2102132736，计算的结果为负数，显然是不合常理的。这是因为在运算过程中，数值超出数据类型的取值范围，Java 语言会按照一定的办法将数据处理为范围之内的一个数值。在该例中，如果将变量 sum 的类型改为 long 型，则会输出正确的答案。

2．语句体忘记加大括号

大括号在 Java 程序中具有非常重要的位置，当语句体的语句不只一条时，必须使用大括号，否则很容易出现错误。例如，下面的语句本意是要计算 1+2+3+…+99+100 的和，但是因为忘记使用大括号将语句体括起来，程序实际上只是重复执行了“sum=sum+i;”语句，从而进入了死循环状态。

```
int sum=0;
int i=1;
while(i<=100)
```

```
sum=sum+i;
i++;
```

为了避免这类错误的发生，最好在任何情况下都使用大括号将语句体括起来。

19.1.4　Java 程序的异常处理机制

在 Java 语言中，当程序执行中发生错误时，错误事件对象可能导致的程序运行错误称为异常（Exception，也称例外），异常会输出错误消息，使其知道该如何正确地处理遇到的问题。

程序运行所导致的异常发生后，怎么处理异常呢？Java 语言提供的异常处理机制，由捕获异常和处理异常两部分组成。

在 Java 程序的执行过程中，如果出现了异常事件，就会生成一个异常对象。生成的异常对象将传递给 Java 运行时系统，这一异常的产生和提交过程称为抛出（throw）异常。当 Java 运行时系统得到一个异常对象时，它将会寻找处理这一异常的代码。找到能够处理这种类型的异常的方法后，运行时系统把当前异常对象交给这个方法进行处理，这一过程称为捕获（catch）异常。如果 Java 运行时系统找不到可以捕获异常的方法，则运行时系统将终止，相应的 Java 程序也将退出。

19.1.5　Java 语言中的异常处理类

Java 通过面向对象的方法进行异常处理，把各种不同的异常进行分类，并提供了良好的接口。这种机制为复杂程序提供了强有力的控制方式。异常处理还有一个好处是在你不能确定和处理异常时，你可以不处理，而把问题提交上去。

Java 中，所有的异常都由类来表示。所有的异常类都是从一个名为 Throwable 的类派生出来的。因此，当程序中发生一个异常时，就会生成一个异常类的某种类型的对象。Throwable 有两个直接子类：Exception 类和 Error 类。

错误 Error 类一般是指与虚拟机相关的问题，如系统崩溃、虚拟机出错误、动态链接失败等，这一类错误无法恢复且不可捕获，将导致应用程序中断。

异常 Exception 类则是指一些可以被捕获且可能恢复的异常情况，如数组下标越界产生异常 ArrayIndexOutOfBoundsExcepton、数值被零除产生异常 ArithmeticException、输入/输出异常 IOException 等。Exception 类和 Error 类都继承于 java.lang.Object 中的 java.lang.Throwable 类，如图 19.6 所示。

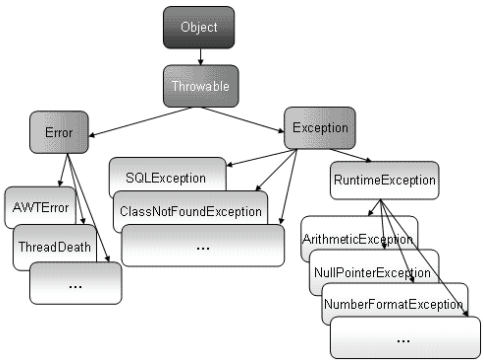


图 19.6　异常类的继承结构

19.2 处理 Java 程序中的异常

Java 的异常处理是通过 5 个关键词来实现的：try、catch、throw、throws 和 finally。一般情况下用 try 来执行一段程序，如果出现异常，系统会抛出（throws）一个异常，这时候可以通过其类型来捕获（catch）它，或最后（finally）由默认处理器来处理。下面是 Java 异常处理程序的基本形式。

19.2.1 使用 try-catch 语句保护代码和捕获异常

异常处理的核心是 try 和 catch，这两个关键字要一起使用。下面是 try/catch 异常处理代码块的基本形式：

```
try{                                          //监视
    //可能发生异常的代码块;
} catch（异常类型异常对象名）{                  //捕获并处理异常
    //异常处理代码块;
}
```



当 try 代码块发生异常并抛出一个异常时，异常会由相应的 catch 语句捕获并处理。如果没有抛出异常，那么 try 代码块就会结束，并且跳过 catch 异常处理代码块。因此，只有在有异常抛出时，才会执行 catch 语句。

下面通过一个简单的示例，说明出现异常时的情况，以及如何监视并捕获一个异常。

【例 19.1】使用 try/catch 进行异常处理的情况。

```
public class Exp5{
    public static void main(String[] args){
        int  i,a;
        try {                                          // 监视一代码块
            i=0;
            a=42/i;
        }catch (ArithmeticException e) {               //捕获一个被零除异常
            System.out.println("被零除");
        }
    }
}
```

该程序输出如图 19.7 所示。

在上面的程序中，进行了一个除法运算，除数为零。这属于运行时异常，在程序编译时没有什么异常信息，但在运行的时候，会发生 ArithmeticException 异常，因此在程序中要捕获它。

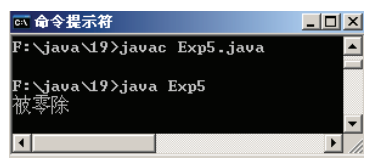


图 19.7 例 19.1 的运行结果

19.2.2 使用多个 catch 子句

由单个代码段可能引起多个异常，处理这种情况时就需要定义两个或更多的 catch 子句，每个子句捕获一种类型的异常。当异常被引发时，每一个 catch 子句被依次检查，第



一个匹配异常类型的子句被执行。当一个 catch 语句执行以后，其他的子句将被忽略。如果没有抛出异常，那么 try 代码块就会结束，并且会跳过它的所有 catch 语句，从最后一个 catch 后面的第一个语句继续执行。其形式如下所示。

```
try{
    //可能发生异常的代码块;
} catch（异常类型 1 异常对象名 1）{
    //异常处理代码块 1;
}
…
catch（异常类型 n 异常对象名 n）{
    //异常处理代码块 n;
}
```

【例 19.2】捕获两种不同类型的异常。

```
public class Exp6{
    public static void main(String[] args){
        try {                                   //监视异常的发生
            int i = args.length;
            System.out.println("i ="+i);
            int j=5/i;
            int k[]={ 1,2,3 };
            k[5]=0;
        }catch(ArithmeticException e) {     //try 块中若发生被 0 除则捕获并处理
            System.out.println("被零除: " + e);
        }catch(ArrayIndexOutOfBoundsException e) {
                                        // try 块中若发生数组越界则捕获并处理
            System.out.println("数组越界异常: " + e);
        }
        System.out.println("执行 catch 块后的语句块");
    }
}
```

该程序运行结果如图 19.8 或图 19.9 所示。

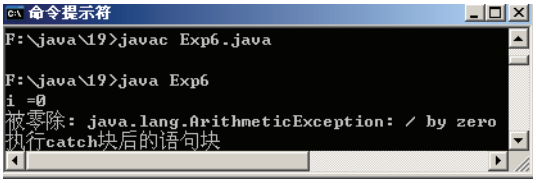


图 19.8　例 19.2 的运行结果（1）



图 19.9　例 19.2 的运行结果（2）

两次运行的不同结果证实，每一个语句只对自己的异常类型做反应。总之，catch 表达式按照在程序中的顺序被检查，只执行匹配的语句，忽略其他所有 catch 代码块。

19.2.3　使用 finally 子句

有时为了确保一段代码不管发生什么异常都要被执行，可以使用关键词 finally 来标出这样一段代码。try/catch/finally 的基本形式如下所示。

```
try{
    //可能发生异常的代码块;
```

```
} catch（异常类型异常对象名）{
    //异常处理代码块;
}
…
finally{
    //无论是否抛出异常都要执行的代码;
}
```

【例 19.3】使用 finally 的示例。

```
public class Exp7{
    public static void main(String[] args){
        try{                        //try 语句，下面的语句可能会发生异常，试着执行
            int [] a=new int[3];        //创建长度为 3 的整型数组
            a[3]=4;                     //下标引用错误
        }catch(ArithmeticException e){
            System.out.println("发生了异常");
        }finally{   //不管 try 中的代码怎样执行，finally 子句中的代码总要执行一次
            System.out.println("最后执行的语句！");
        }
    }
}
```

该程序运行结果如图 19.10 所示。

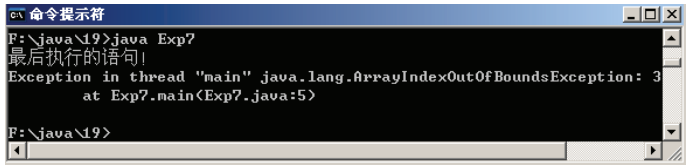


图 19.10　例 19.3 的运行结果

在通常的执行过程中，控制到达 try 块的末尾之后，继续执行 finally 块，该块主要执行必要的清除工作。如果由于 return、continue、break 等语句而使控制转移到 try 块之外，在控制转移到 try 块外面之前，仍然执行 finally 块中的内容。如果在 try 块中出现了异常，并且在方法中指定了一个 catch 块来处理该异常，控制将首先转移到 catch 块，然后转移到 finally 块。如果没有 catch 块来处理该异常，控制将首先转换到 finally 块，然后上移至以前的方法调用，直到处理该异常为止。但在 try 块或 catch 块中如果执行了 System.exit()方法，则终止虚拟机，此时不会再执行 finally 块。

19.2.4　可嵌入的 try 块

一个 try 代码块可以嵌入到另一个 try 代码块当中。由于内部 try 代码块产生的异常如果没有被与该内部 try 代码块相关的 catch 捕获，就会传到外部 try 代码块。

【例 19.4】嵌套 try 语句的示例。

```
public class Exp8{
    public static void main(String args[]){
        int data1[]={2,4,6,8,10,12};
        int data2[]={1,0,2,4,3};
        try{                                          //外部 try 代码块
            for(int i=0;i<data1.length;i++) {
```

```
                    try{                        //内部 try 代码块
                        System.out.println(data1[i]+ "/"+data2[i]+ "is"+data1
[i]/data2[i]);
                    }catch(ArithmeticException e){
                                                //捕获并处理内部 try 代码块产生的异常
                        System.out.println("不能被零除！");
                    }
                }
            }catch(ArrayIndexOutOfBoundsException e){
                                                //捕获并处理内部 try 代码块产生的异常
                System.out.println("程序被终止！");
            }
        }
    }
```

该程序运行结果如图 19.11 所示。



图 19.11　例 19.4 的运行结果

在本例中，异常由内部 try 处理，因此允许程序继续执行。而由外部 try 捕获的数组越界错误使程序终止。通常嵌入式 try 代码块用于以不同方式处理不同类型的错误。某些类型的错误是致命的，无法修改。某些错误则较轻，可以马上处理。许多程序员在使用外部 try 代码块捕获大部分严重错误的同时，让内部 try 代码处理不太严重的错误。

19.3　抛出异常

对于处理不了的异常或者要转型的异常，在方法的声明处通过 throws 语句声明抛出异常，而 throw 语句用于在方法体内抛出一个异常。

19.3.1　使用 throws 声明可能会抛出异常的方法

如果在一个方法中生成了一个异常，但是这一方法并不确切地知道该如何对这一异常事件进行处理，这时，一个方法就应该声明抛出异常，使得异常对象可以从调用向后传播，直到有合适的方法捕获它为止。

声明抛出异常是在一个方法声明中的 throws 子句中指明的。下面是包含 throws 子句的方法的基本形式。

```
[修饰符] 返回类型 方法名（参数 1,参数 2,……）throws 异常列表{
    ……
}
```

例如，下面的代码声明了一个 read()方法，该方法可能会抛出 IOException 异常。

```
public int read () throws IOException{
    ……
}
```

throws 子句中同时可以指明多个异常，说明该方法将不对这些异常进行处理，只是声明抛出它们，由调用此方法的其他方法负责捕获和处理。例如，下面的方法声明抛出 IOException 异常和 IndexOutOfBoundsException 异常，而不是在方法内部处理这些异常。

```
    public static void main(String args[]) throws IOException, IndexOutOf
BoundsException{
        ……
    }
```

【例 19.5】声明抛出异常的程序。

```
    public class Exp9{
        static void throwOne() throws IllegalAccessException{       //声明抛出
IllegalAccessException 类异常
            System.out.println("在 throwOne 中.");
            throw new IllegalAccessException("非法访问异常");
                                              //抛出 IllegalAccessException 类异常
        }
        public static void main(String args[]){
            try{                                                    //监视异常的发生
                throwOne();
            }catch(IllegalAccessException e){                       //捕获并处理异常
                System.out.println("捕获"+e);
            }
        }
    }
```

该程序运行结果如图 19.12 所示。

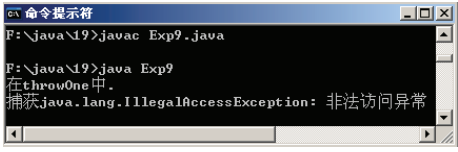


图 19.12　例 19.5 的运行结果

在本例中，定义了一个静态的方法 throwOne()，在方法的头部声明中，使用关键字 throws 声明该方法有可能抛出 IllegalAccessException 类异常，因此任何调用 throwOne()方法的对象，都要处理这个异常。

19.3.2　使用 throw 抛出异常

前面的例子中所涉及的异常都是由 Java 虚拟机（JVM）自动产生的。抛出异常首先要生成异常对象，异常或者由虚拟机生成，或者由某些类的实例生成，也可以在程序中生成。手动抛出异常对象是通过 throw 语句实现的，但可以抛出的异常必须是 Throwable 或其子类的实例。其基本形式如下。

```
throw 异常名;
```

throw 关键字主要用在 try 块中，用来说明已经发生的异常情况。throw 关键字后面跟随一个从类 Throwable 中派生的异常对象，用来说明抛出的异常类型。

【例 19.6】使用 throw 关键字抛出 IOException 异常。

```
import java.io.*;
public class Exp10{
    public static void main(String [] args) {
        try{
            System.out.println("…正在运行程序…");
            throw new IOException("用户自行产生异常");
                                              //抛出 IOException 类异常
        }catch(IOException e){                          //捕获并处理异常
            System.out.println("已捕获了该异常！");
```

```
        }
    }
}
```

该程序运行结果如图 19.13 所示。

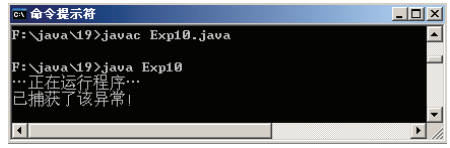


图 19.13　例 19.6 的运行结果

在这个程序示例中，使用 throw 抛出一个 IOException 类型的异常。只有使用 throw 才会真正抛出异常，而前面讲的关键字 throws 仅是用于方法声明，说明“可能”会产生异常。



注意在 throw 语句中是如何使用 new 创建 IOException 的。这里 throw 抛出一个对象，所以必须为类型创建一个对象来抛出。

19.3.3　异常类常用方法

Exception 类自己没有定义任何方法，它继承了 Throwable 提供的一些方法。因此，所有异常，包括自己创建的自定义异常，都可以获得 Throwable 定义的方法。同时我们还可以在创建的自定义异常类中覆盖一个或多个这样的方法。表 19.1 列出了三种常用的方法。

表 19.1 异常类常用方法

方法声明	方法作用
public String getMessage()	getMessage()方法返回描述当前异常类的消息字符串
public String　toString()	toString()方法返回描述当前异常类的消息的字符串，一般由三部分组成：此对象实际类的名称、冒号和空格、此对象 getMessage()方法的结果
public void printStackTrace()	printStackTrace()方法没有返回值，它的功能是完成一个打印操作，在当前的标准输出上打印输出当前异常对象的堆栈使用轨迹，即程序先后调用执行了哪些对象或类的哪些方法，使得运行过程中产生了这个异常对象

19.4　自定义异常

尽管 Java 的内置异常能够处理大多数常见错误，但有时还可能出现系统没有考虑到的异常，因此我们仍然希望建立自己的异常类型，来处理所遇到的特殊情况：通过继承 Exception 类或它的子类，实现自定义异常类。对于自定义异常，必须采用 throw 语句抛出异常，这种类型的异常不会自行产生。

19.4.1　创建自定义异常类

使用 Java 内置的异常类可以描述在编程时出现的大部分异常情况。除此之外，用户还可以自定义异常。所有的用户自定义异常类都必须由 Exception 类或者 Exception 类的子类派生，所以必须显示指明异常类的基类。自定义异常的基本形式如下所示。

```
class 自定义异常 extends 父异常类名{
    类体;
}
```

【例 19.7】自定义异常示例。

```
class Exp extends Exception{                          //自定义异常类 Exp
```

```
    private int show;
    Exp(int a){                                          //自定义异常类 Exp 构造方法
        show = a;
    }
    public String toString(){
        return "Exp<" + show + ">";
    }
}
public class Exp11{                                      //测试自定义异常类 Exp
    static void caculate(int a) throws Exp{              //声明抛出自定义异常类 Exp
        System.out.println("对["+ a +"]已经进行过相应的操作");
        if(a>100)
            throw new Exp (a);                           //抛出自定义异常类 Exp
        System.out.println("执行该算法正常退出！");
    }
    public static void main(String args[]){
        try{                                             //监视
            caculate(1);
            caculate(1000);
        }catch (Exp e){                                  //捕获并处理自定义异常
            System.out.println("捕获了异常" + e);
        }
    }
}
```



图 19.14　例 19.7 的运行结果

该程序运行结果如图 19.14 所示。

使用自定义异常可以归纳为如下几点：

- 自定义异常类必须是 Throwable 的直接或间接子类。
- 一个方法所声明抛出的异常是作为这个方法与外界交互的一部分而存在的。方法的调用者必须了解这些异常，并确定如何正确地处理它们。
- 用异常代表错误，而不要再使用方法返回值。

19.4.2　使用 throw 和 try-catch 语句处理自定义异常

在程序中使用自定义异常类，大体可分为以下几个步骤：

① 创建自定义异常类。

② 在方法中通过 throw 关键字抛出异常对象。

③ 如果在当前抛出异常的方法中处理异常，可以使用 try-catch 语句捕获并处理；否则在方法的声明处通过 throws 关键字指明要抛出给方法调用者的异常，继续进行下一步操作。

④ 在出现异常方法的调用者中捕获并处理异常。

【例 19.8】使用自定义异常示例。

```
import java.util.*;
public class Exp12 {
    static int avg(int number1, int number2) throws MyException {
//声明抛出自定义异常
        if (number1< 0 || number2< 0) {
```

```
            throw new MyException("不可以使用负数");        //抛出自定义异常
        }
        if (number1> 100 || number2> 100) {
            throw new MyException("数值太大了");           //抛出自定义异常
        }
        return (number1+ number2)/ 2;                      //返回语句
    }
    public static void main(String[] args) {
        System.out.println("求 2 个数的平均数！" +"请输入两个数，要求是都小于 100
的数：");
        Scanner in=new Scanner(System.in); //创建一个对象，用于读取用户输入
        int number1=in.nextInt();          //从键盘获得输入
        int number2=in.nextInt();          //从键盘获得输入
        try {
            int result = avg(number1, number2);        //调用方法 avg()
            System.out.println(result);
        }catch (MyException e) {                       //捕获自定义异常
            System.out.println(e) ;                    //打印自定义异常信息
        }
    }
}
class MyException extends Exception {          //自定义异常继承 Exception 类
    MyException(String ErrorMessage) {
        super(ErrorMessage);
    }
}
```

该程序运行结果如图 19.15 所示。



图 19.15　例 19.8 的运行结果

在上面的程序中，自定义了一个异常类 MyException，它是 Exception 的子类。在方法 avg() 的定义中，使用关键字 throws 声明该方法可能会抛出 MyException 类型的异常，并且在 avg()方法内部，当一定的条件满足时（“number1<0||number2<0”或“number1>100||number2>100”），使用关键字 throw 创建一个 MyException 类型的异常对象并抛出。在 main()方法中调用了 avg()方法，所以必须处理 avg() 方法可能抛出的异常。在这里为使用 catch 语句捕获它并输出异常信息。

19.5　小结

本章主要向读者介绍了 Java 中的异常处理机制。异常处理机制是保证 Java 程序正常运行、具有较高安全性的重要手段。对于开发良好的编程习惯是非常重要的。读者在学习本章内容时，把重点放在对 Java 程序中的异常处理上。而在学习到自定义异常时，读者可能会感觉到有些困难。建议将自定义异常作为了解内容即可。下一章将向读者介绍 Java 程序中的输入和输出方法。

第20章 输入与输出

输入/输出是指应用程序与外部设备及其他计算机进行数据交流的操作，如读写硬盘数据、向显示器输出数据、通过网络读取其他结点的数据等。任何一种编程语言必须拥有输入输出的处理方式，Java 语言也不例外。Java 语言的输入/输出数据是以流的形式出现的，并且 Java 提供了大量的类对流进行操作，从而实现了输入/输出功能。

20.1 文件类（File 类）的使用

在正式了解 Java 中如何处理输入/输出之前，要先了解一下在 Java 中如何进行文件和目录的管理。文件是用来保存数据的，目录是管理文件的特殊机制，同类文件保存在同一目录下可以简化文件管理。因此，掌握文件和目录的操作对于编程员是十分重要的。

在 Java 语言中，文件和目录的管理是由 java.io.File 类来实现的。File 类也属于 java.io 包，但它不是 InputStream 或者 OutputStream 的子类，因此不负责数据的输入和输出，而是专门管理磁盘的文件和目录。

每个 File 类的对象表示一个磁盘文件或目录，其对象属性中包含了文件或目录的相关信息，如名称、长度、所含文件个数等，调用它的方法则可以完成对文件或目录的常用管理操作，例如，创建、删除等操作。

1．创建 File 类对象

每个 File 类的对象都对应系统的一个磁盘文件或目录，所以创建 File 类对象时需指明它所对应的文件或目录名。File 类提供三种格式以不同的参数形式灵活地接收文件和目录名信息。

- File 对象名=new File(String path)；
- File 对象名=new File(String Path，String name)；
- File 对象名=new File(File dir，String name)。



参数 path 指明了新创建的 File 对象对应的磁盘文件或目录名及其路径名，path 参数也可以对应磁盘上的某个目录；参数 name 表示文件或目录名；参数 dir 为另一个已经存在的代表某磁盘目录的 File 对象，表示文件或目录的路径。

.2．文件或目录的属性

经常使用 File 类的下列方法获取文件本身的一些信息：

- public String getName()，获取文件或目录的名字。
- public boolean canRead()，获取文件的可读属性，如果为可读文件，返回 true。
- public boolean canWrite()，获取文件的可写属性，如果为可写文件，返回 true。
- public boolean exists()，判断文件或目录是否存在，如果存在则返回 true，否则返回 false。
- public long length()，获取文件的长度（单位是字节）。
- public String getAbsolutePath()，获取文件的绝对路径 “c:/file/hello.txt”。
- public String getParent()，获取文件的父目录 “c:/file”。
- public boolean isFile()，判断是否为文件，如果对象代表有效文件，则返回 true。
- public boolean isDirectroy()，判断是否为目录，如果对象代表有效目录，则返回 true。
- public boolean isHidden()，判断文件是否是隐藏文件。
- public long lastModified()，获取文件最后修改的时间（时间是从 1970 年午夜至文件最后修改时刻的毫秒数）。
- public String[] list()，用字符串形式返回目录下的全部文件。
- public File[] listFiles()，用 File 对象形式返回目录下的全部文件。

在下面的例子中，我们使用上述的一些方法，获取某些文件信息。

【例 20.1】获取文件或目录的基本信息。

```
import java.io.File;
import java.util.Date;
public class GetFileInfos{
    public static void println(String s){
        System.out.println(s);
    }
    public static void main(String[] args){
        //用文件路径新建一个文件对象。路径可以是绝对路径，也可以是相对路径
        File file = new File("D:/work/newTemp.txt");
        println("文件名： " + file.getName());  //获取文件的名字，不包括路径
        println("文件路径： " + file.getPath());//获取文件的绝对路径
        println("绝对路径： " + file.getAbsolutePath()); //获取文件的绝对路径
        println("父目录： " + file.getParent());//获取抽象路径名的父抽象路径名
        println("文件是否存在： " + file.exists());
        println("是否可读： " + file.canRead());
        println("是否可写： " + file.canWrite());
        println("是否是隐藏文件： " + file.isHidden());
        println("是否是普通文件： " + file.isFile());
        println("是否是目录： " + file.isDirectory());
        println("文件路径是否是绝对路径： " + file.isAbsolute());
        println("文件路径的 URI: " + file.toURI());
        println("文件最后修改时间： " + new Date(file.lastModified()));
        println("文件大小： " + file.length() + "bytes");
        println("是否可执行："+file.canExecute());
    }
}
```

在这个程序示例中，因为要对文件进行操作，所以首先导入 java.io.File 类。在构造 File 对象时，将文件的路径作为参数传递进来，在本程序中为“D:/work/newTemp.txt”。然后调用 File 对象所提供的各个方法来获得文件对象的信息。

20.2 流的概念

Java 语言的输入/输出功能是十分强大而灵活的，美中不足的是看上去输入/输出的代码并不是很简洁，因为你往往需要包装许多不同的对象。在 Java 类库中，I/O 部分的内容是很庞大的，因为它涉及的领域很广泛：标准输入/输出、文件的操作、网络上的数据流、字符串流、对象流等。本节的目的是对流做一个简要的介绍。

20.2.1 流的定义

所谓流，是指同一台计算机或网络中不同计算机之间有序运动着的数据序列，Java 把这些不同来源和目标的数据都统一抽象为数据流。流序列中的数据可以是没有进行加工的原始数据（二进制字节数据），也可以是经过编码的符合某种格式规定的数据，Java 中提供了不同的流类对它们进行处理。

20.2.2 输入流与输出流

一个流，必有源端和目的端，它们可以是计算机内存的某些区域，也可以是磁盘文件，甚至可以是 Internet 上的某个 URL。流的方向是重要的，根据流的方向，流可分为两类：输入流和输出流。用户可以从输入流中读取信息，但不能写它。相反，对输出流，只能往输出流写，而不能读它。

1．输入流

在 Java 中，程序可以打开一个输入流，输入流的信息源可以位于文件、内存或网络套接字(socket)等地方，信息源的类型可以是包括对象、字符、图像、声音在内的任何类型。一旦打开输入流后，程序就可从输入流串行地读数据。从输入流读数据的一般过程如图 20.1 所示。

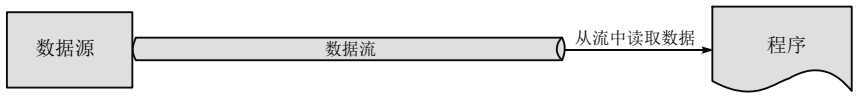


图 20.1　输入流读取数据

2．输出流

类似地，程序也能通过打开一个输出流并按顺序写入数据来将信息送至目的端。输出流写数据的一般过程如图 20.2 所示。

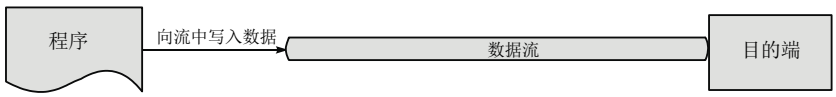


图 20.2　输出流写入数据

20.2.3　字节流与字符流

Java 流包括字节流和字符流，字节流通过 IO 设备以字节数据的方式读入，而字符流则通过字节流读入数据转化成字符“流”的形式由用户驱使。

20.3　字节流

对字节流，其流类的层次结构如图 20.3 和图 20.4 所示，从图中可以看出，InputStream 是所有字节输入流的祖先，而 OutputStream 是所有字节输出流的祖先。

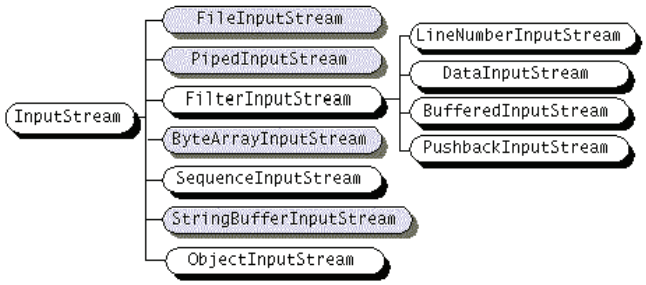


图 20.3　字节输入流结构图

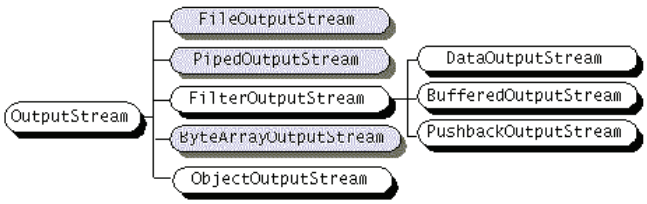


图 20.4　字节输出流结构图

20.3.1　InputStream 类与 OutputStream 类

在 Java I/O 流中，所有对字节流进行处理的类，都继承自 InputStream 类和 OutputStream 类，这是两个抽象类。

1．InputStream 类

InputStream 类是个抽象类，作为字节输入流的直接或间接的父类，它定义了许多有用的、所有子类必需的方法，包括读取、移动指针、标记、复位、关闭等方法。下面介绍了这些方法，需要注意的是，这些方法大多可能抛出 IOException 异常。

- int read()：读取一个字节，返回值为所读的字节。
- int read(byte b[])：读取多个字节，放置到字节数组 b 中，通常读取的字节数量为 b 的长度，返回值为实际读取的字节的数量。
- int read(byte b[], int off, int len)：读取 len 个字节，放置到以下标 off 开始的字节数组 b 中，返回值为实际读取的字节的数量。
- int available()：返回值为流中尚未读取的字节的数量。
- long skip(long n)：读指针跳过 *n* 个字节不读，返回值为实际跳过的字节数量。

- close()：流操作完毕后必须关闭。
- void mark(int readlimit)：记录当前读指针所在位置，readlimit 表示读指针读出 readlimit 个字节后所标记的指针位置才失效。
- void reset()：把读指针重新指向用 mark 方法所记录的位置。
- boolean markSupported()：当前的流是否支持读指针的记录功能。

2．OutputStream 类

OutputStream 类也是抽象类，作为字节输出流的直接或间接的父类，当程序需要向外部设备输出数据时，需要创建 OutputStream 的某一个子类的对象来完成。以下介绍该类的常用方法，与 InputStream 类似，这些方法也可能抛出 IOException 异常。

- void write(int b)：往流中写一个字节 b。
- void write(byte b[])：往流中写一个字节数组 b。
- void write(byte b[], int off, int len)：把字节数组 b 中从下标 off 开始，长度为 len 的字节写入流中。
- flush()：把缓存中所有内容强制输出到流中。
- close()：流操作完毕后必须关闭。

20.3.2 文件输入流 FileInputStream 类与文件输出流 FileOutputStream 类

FileInputStream 类和 FileOutputStream 类中，第一个类的源端和第二个类的目的端都是磁盘文件，它们的构造方法允许通过文件的路径名来构造相应的流。如：

- FileInputStream infile = new FileInputStream("myfile.dat");
- FileOutputStream outfile = new FileOutputStream("results.dat");



构造 FileInputStream 对象时,对应的文件必须存在并且是可读的，而构造 FileOutputStream 对象时，如输出文件已存在，则必须是可覆盖的。

1．FileInputStream 类

如果用户的文件读取需求比较简单，那么用户可以使用 FileInputStream 类，该类是从 InputStream 中派生出来的简单输入类，该类的所有方法都是从 InputStream 类继承来的。

【例 20.2】使用文件输入流读取文件。

```
import java.io.FileInputStream;
import java.io.IOException;
public class ReadFileDemo {
     public static void main(String[] args) {
          try {
               FileInputStream in = new FileInputStream("TestFile.txt");
// 创建文件输入流对象
               int n = 512;                          //设定读取的字节数
               byte buffer[] = new byte[n];          //读取输入流
               //读取 n 个字节，放置到以下标 0 开始的字节数组 buffer 中，返回值为实际读取的字节的数量
               while ((in.read(buffer, 0, n) != -1) && (n > 0)) {
                    System.out.print(new String(buffer));
               }
```



```
            System.out.println();
            in.close();              //关闭输入流
        } catch (IOException ioe) {
            System.out.println(ioe);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

本例以 FileInputStream 的 read(buffer)方法，每次从源程序文件 TestFile.txt 中读取 512 个字节，存储在缓冲区 buffer 中，再将以 buffer 中的值构造的字符串 new String(buffer)显示在屏幕上。



使用文件输入流构造器建立通往文件的输入流时，可能会出现错误（也被称为异常），例如，要打开的文件可能不存在。当出现 I/O 错误时，Java 生成一个出错信号，它使用一个 IOException 对象来表示这个出错信号。程序必须使用一个 try-catch 块检测并处理这异常。

2．FileOutputStrearn 类

FileOutputStream 提供了基本的文件写入能力。除了从 OutputStream 类继承来的方法以外，FileOutputStream 类还有两个构造器：

- FileOutputStream(String name)。
- FileOutputStream(File file)。

第一个构造器使用给定的文件名 name 创建一个 FileOutputStream 对象。第二个构造器使用 File 对象创建 FileOutputStream 对象。该类可以使用 write 方法把字节发送给输出流。

下例用 System.in.read(buffer)从键盘输入一行字符，存储在缓冲区 buffer 中，再以 FileOutputStream 的 write(buffer)方法，将 buffer 中的内容写入文件 WriteFile.txt 中。

【例 20.3】使用文件输出流写入文件。

```
import java.io.FileOutputStream;
import java.io.IOException;
public class WriteFileDemo{
    public static void main(String[] args) {
        try {
            System.out.print("输入要保存文件的内容：");
            int count, n = 512;
            byte[] buffer = new byte[n];     //定义存放读入信息的字节数组
            count = System.in.read(buffer); //读取标准输入流
           FileOutputStream os = new FileOutputStream("WriteFile.txt");
                                          //创建文件输出流对象
            os.write(buffer, 0, count);
             //把字节数组 buffer 中从下标 0 开始，长度为 count 的字节写入流中
            os.close();                   //关闭输出流
            System.out.println("已保存到 WriteFile.txt!");
        } catch (IOException ioe) {       //捕获 IOException 异常
            System.out.println(ioe);      //输出异常信息
        } catch (Exception e) {           //捕获其他异常
            System.out.println(e);        //输出其他异常信息
```

```
            }
        }
    }
```

在这个程序中，使用“System.on.read(buffer);”语句时，程序会暂停，等待用户输入内容。用户输入的内容会保存在字节数组 buffer 中，并返回读入的内容的长度保存到变量 count 中。因为对文件的读写有可能发生 IO 异常，所以在程序中要处理 IOException 异常。编译并运行此程序，会在同一目录下，提示输入要保存文件的内容。输入一段话，按回车确认，这时会在目录下生成一个名为 WriteFile.txt 的文本文件。打开这个文本文件，会看到刚刚输入的这段话。

20.3.3 缓存输入流 BufferedInputStream 类与缓存输出流 BufferedOutputStream 类

从字节流的 stream 类层次结构图看出，BufferedInputStream 类为 FilterInputStream 的子类,而 FilterInputStream 又是 InputStream 的子类。从类名中，可以了解这个类主要在流的输入过程中提供了缓存的功能。也可以从其构造函数中进一步地了解到其是跟其他流类一起搭配使用的。BufferedOutputStream 类跟 BufferedInputStream 类一样，不同的是一个用于读数据一个用于写数据而已。

java.io.BufferedInputStream 与 java.io.BufferedOutputStream 可以为 InputStream、OutputStream 类的对象增加缓冲区功能，构建 BufferedInputStream 实例时，需要给定一个 InputStream 类型的实例，实现 BufferedInputStream 时，实际上最后是实现 InputStream 实例。同样，在构建 BufferedOutputStream 时，也需要给定一个 OutputStream 实例，实现 BufferedOutputStream 时，实际上最后是实现 OutputStream 实例。

【例 20.4】使用缓存输入、输出流读写文件。

```
import java.io.*;
public class BufferedStreamDemo {
    public static void main(String[] args) {
        try {
            FileInputStream in = new FileInputStream("D:/b.txt");
                                                    //来源文件
            FileOutputStream out = new FileOutputStream("C:/a.txt");
                                                    //目的文件
            BufferedInputStream bufferedIn = new BufferedInputStream(in);
                                                    //缓冲输入流
            BufferedOutputStream bufferedOut = new BufferedOutputStream(out);
                                                    //缓冲输出流
            byte[] data = new byte[1];              //接收数据的字节数组
            while (bufferedIn.read(data) != -1) {
                bufferedOut.write(data);            //将缓冲区中的数据全部写出
            }
            bufferedOut.flush();                    //刷新缓冲输出流
            bufferedIn.close();                     //关闭输入流
            bufferedOut.close();                    //关闭输入流
        } catch (ArrayIndexOutOfBoundsException e) {  //捕获数组越界异常
            e.printStackTrace();
        } catch (IOException e) {                       //捕获 IO 异常
```

```
                e.printStackTrace();
            }
        }
    }
```

在这个程序中，构建了一个文件输入流对象 in 和一个文件输出流对象 out。同时为了提高数据的批量输入/输出速度，对 in 和 out 分别加上缓冲输入流和缓冲输出流，然后通过缓冲输入流和输出流对数据进行读写。因为读入的数据是保存在字节数组 data 中的，所以在进行文件读写时，有可能发生数组下标越界的异常，因此需要在程序中处理 ArrayIndexOutOfBoundsException 异常和 IOException 异常。



为了确保缓冲区中的数据一定被写出至目的地，建议最后执行 flush()将缓冲区中的数据全部写出至目的流中。



20.4　字符流

字符流主要是用于支持 Unicode 的文字内容，绝大多数在字节流中所提供的类，都可在此找到对应的类。其中输入流 Reader 抽象类帮助用户在 Unicode 流内获得字符数据；而 Writer 类则是实现了输出。可以利用 Reader 类来读取由 Writer 写入的流。对于字符流，其 stream 类的层次结构如图 20.4 和图 20.5 所示。

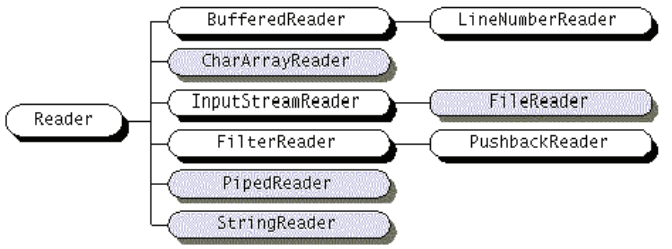


图 20.4　字符输入流结构图

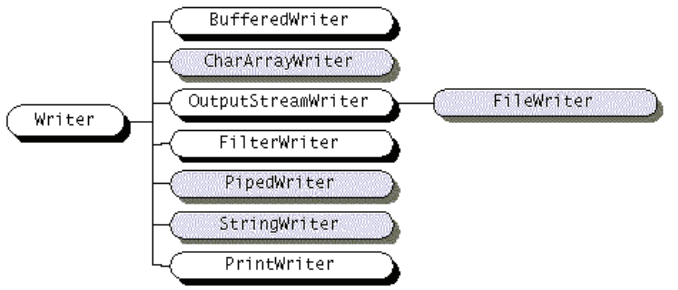


图 20.4　字符输出流结构图

从图中可以看出，Reader 类是所有字符输入流的祖先，而 Writer 类是所有字符输出流的祖先。

20.4.1　Reader 类和 Writer 类

前面说过，在 JDK1.1 之前，java.io 包中的流只有普通的字节流（以 byte 为基本处理

单位的流），这种流对于以 16 位的 Unicode 码表示的字符流处理很不方便。从 JDK1.1 开始，java.io 包中加入了专门用于字符流处理的类，它们是以 Reader 和 Writer 为基础派生的一系列类。同类 InputStream 和 OutputStream 一样，Reader 和 Writer 也是抽象类，只提供了一系列用于字符流处理的接口。它们的方法与类 InputStream 和 OutputStream 类似，只不过其中的参数换成字符或字符数组。

20.4.2 字符文件读写器 FileReader 类和 FileWriter 类

FileReader 类和 FileWriter 类分别是 Reader 和 Writer 的子类，其构造方法分别是：

- FileReader(String filename)；
- FileReader(File f)；
- FileWriter(String filename)；
- FileWriter(File f)。

前面的 FileInputStream 使用字节读取文件，字节流不能直接操作 Unicode 字符，所以 Java 提供了字符流。由于汉字在文件中占用 2 个字节，如果使用字节流，读取不当会出现乱码现象，采用字符流就可以避免这个现象，因为在 Unicode 字符中，一个汉字被看做一个字符。

1．FileReader 类

FileReader 类用于读取文件，每次读取文件中第一个未读取过的字符，并以 ASCII 码或者 UTF-8 码的形式输入到程序中。创建 FileReader 类对象的格式为：

```
FileReader fr=new FileReader(filename);
```

其中文件名必须是文件的完整路径和文件名，如果程序和该文件保存在同一目录下，则可以只用文件名而不需要其路径。

FileReader 类中的 read()方法用来读取字符并返回一个相应的 int 类型数据。当读到文件的结尾处时，则返回数值-1。在完成文件数据的读取后需要使用 close()方法关闭打开的文件。

【例 20.5】读取文件 student.txt 的内容显示在屏幕上。

```
import java.io.*;
public class FileReadDemo{
     public static void main(String[] args) throws IOException{
          FileReader fr=new FileReader("student.txt");
                                              //创建一个 FileReader 类型的对象
          int c=fr.read();                    //从文件中读取字符并存入 c 变量中
          while(c!=-1){                       //判断文件内容是否结束
               System.out.print((char)c);     //输出读取的字符到控制台
               c = fr.read();                 //读取下一个字符
          }
          fr.close();                         //关闭文件阅读器
     }
}
```

在这个程序中，构建了一个文件阅读器 FileReader 的对象实例 fr，通过 while 循环语句调用其 read()方法依次读取一个字符，并判断是否到了文件结束处。如果没有到文件结尾，

则将读取的整数强制类型转换为 char 类型，并输出到控制台中。在程序最后，要关闭流。



在声明 main()方法时，添加了 throws IOException 用来处理输入和输出文件时发生的异常。当然，也可以使用 try 和 catch 语句来处理异常。但是如果不需要异常的具体处理办法，则可以只用 throws 关键字。

2．FileWriter 类

FileWriter 类用于将数据写入文件，创建 FileWriter 类对象的格式为：

```
FileWriter fr=new FileWriter (filename);
```

其中，文件名必须是文件的完整路径和文件名，如果程序和该文件保存在同一目录下，则可以只用文件名而不需要其路径。如果该文件名不存在，则系统会自动创建该文件。FileWriter 类中的 write()方法可以将字符或者字符串写入文件中。当完成数据写入操作后，使用 close()方法关闭文件。

【例 20.6】创建一个文件 student.txt 并写入一些字符串。

```
import java.io.*;
public class FileWriteDemo{
     public static void main(String[] args) throws IOException{
          FileWriter fw = new FileWriter("student.txt");
                                                  //创建一个 FileWriter 类型的对象
          fw.write("This is my ");                //向文件中写入字符串
          fw.write("stu");
          fw.write("dent");
          fw.write('.');                          //向文件中写入字符
          fw.write("txt");
          fw.write("   这是我的文件。");
          fw.close();                             //关闭流
     }
}
```

在这个程序中，构建了一个文件写入器 FileWriter 的对象实例 fw，调用其 write()方法向文本文件 student.txt 中写入各种文本。在这个程序中，没有在代码中捕获异常，而是在 main()方法头部使用 throws 关键字抛出 IOException 异常，将异常交给虚拟机来处理。

20.4.3　带缓存的字符文件读写器 BufferedReader 类和 BufferedWriter 类

一般情况下，为了提高字符文件读写效率，通常需要为文件读写器添加一个缓冲读写器，分别为 BufferedReader 类和 BufferedWriter 类。

1．BufferedReader 类

假如上面的例子使用的文件 Student.txt 是一个学生名单，每个姓名占一行。如果我们想读取名字，那么每次必须读取一行，但 FileReader 类没有提供这种方法，所以我们必须把这个流(对象)再接到另外一个流上，从后一个流中读取名单。Java 提供了名为 BufferedReader 的类，主要是用来实现读取文件中的一个段落，其格式为：

```
BufferedReader  br=new  BufferedReader(new FileReader(filename));
```

也可分开写成：

```
FileReader  tofile=new FileWriter(filename);
BufferedReader br=new BufferedWriter(tofile);
```

在声明了 BufferedReader 类的对象后，就可以调用其 readLine()方法来读取文件中的数据。当读取到回车符(\n)时，表示本次读取结束，将所读取到的内容以字符串数据的形式输入到程序中，下次读取从回车符后面的数据开始。当所有的内容都读取完后，返回值为 null。在完成文件数据的读取后需要使用 close()方法关闭打开的文件。

【例 20.7】用 BufferedReader 类来读取文件 Student.txt 中的内容并显示在屏幕上。

```
import java.io.*;
public class BufferedReaderDemo{
     public static void main(String[] args) throws IOException{
          FileReader fr = new FileReader("student.txt");
          BufferedReader  br = new  BufferedReader(new FileReader("student.
txt"));
          String  s = br.readLine();//创建变量 s 用于存储从文件中读取的第一行数据
          while(s! = null){         //判断 s 变量是否接收到数据
               System.out.print(s+"\n");
               S = br.readLine();   //读取下一行数据并存储到 s 中
          }
          br.close();               //关闭流
     }
}
```

因为使用 BufferedReader 类读取文件时，回车符号(\n)不会作为数据输入到程序中，所以需要打印添加该符号来保持输出内容与文件中的内容完全一样。同样，我们也可以用 BufferedReader 类来实现从键盘读取字符或字符串。

【例 20.8】用 BufferedReader 类来实现从键盘读取字符。

```
import java.io.*;
class KeyboardDemo1{
     public static void main(String[] args) throws IOException{
          char a;                    //声明一个 char 类型的变量
          BufferedReader br = new BufferedReader(new InputStreamReader
(System. in));
          a = (char) br.read();      //从输入流 br 中读取字符并存入字符变量 a 中
          System.out.println(a);
     }
}
```

【例 20.9】用 BufferedReader 类来实现从键盘读取字符串。

```
import java.io.*;
public class KeyboardDemo2{
     public static void main(String args[]) throws IOException{
          BufferedReader in = new BufferedReader(new InputStreamReader
(System. in));
          String a;
          System.out.print("请输入任一字符串：");
          a = in.readLine();    //从输入流 br 中读取字符串并存入字符串变量 a 中
          System.out.println("你输入的字符串是："+a);
          in.close();
     }
}
```



在这两个程序示例中，演示了 BufferedReader 的两个方法：read()和 readLine()。其中 read()方法一次读取一个字符，而 readLine()方法一次可以读取一行字符。

2．BufferedWriter 类

类似地，可以将 BufferedWriter 流和 FileWriter 流连接在一起，然后使用 BufferedWriter 流将数据写到目的地，创建 BufferedWriter 类对象的格式为。例如：

```
BufferedWriter  br=new  BufferedWriter (newFileWriter (filename));
```

也可像下面这样分开写：

```
FileWriter  fromfile=new FileWriter(filename);
BufferedWriter bw=new BufferedWriter(fromfile);
```

在声明 BufferedWriter 类的对象后，调用其 newLine()方法来写入一个回车符号。因为不同的操作系统平台，其回车符号的表达方式不同，所以，可以使用 newLine()方法来直接产生系统声明的回车符号，而不必在意其具体的表达方式。例如，下面的程序创建一个文件 student.txt 并写入字符串和回车符号。

【例 20.10】创建一个文件 student.txt 并用 BufferedWriter 流将字符串和回车符号写入其中。

```
import java.io.*;
public class BufferedWriterDemo{
     public static void main(String[] args) throws IOException{
          BufferedWriter  bw=new  BufferedWriter(new  FileWriter("student.
txt"));
          bw.write("This is my student.txt");   //向文件中写入字符串
          bw.newLine();                         //换行
          bw.write("这是我的文件。");             //向文件中写入内容
          bw.close();                           //关闭流
     }
}
```

20.4.4 打印流 PrintStream 类和 PrintWriter 类

要想输入和输出各种数值类型，通常使用打印输入流 PrintStream 和 PrintWriter。其中 PrintStream 操纵的是字节，而 PrintWriter 操纵的是字符。

1．PrintStream 类

PrintStream 为其他输出流添加了功能，使它们能够方便地打印各种数据值表示形式。与其他输出流不同，PrintStream 永远不会抛出 IOException；另外，为了自动刷新，可以创建一个 PrintStream；这意味着可在写入 byte 数组之后自动调用 flush()方法，可调用其中一个 println()方法，或写入一个换行符或字节('\n')。PrintStream 是向标准输出设备的输出流，可直接输出各种类型的数据。其构造函数为：

```
public PrintStream(OutputStream out)           //创建一无 flush 的标准输出流
public PrintStream(OutputStream out,boolean autoFlush) //创建标准输出流
```

【例 20.11】随机产生 10 个数，使用 PrintStream 流将它们放入 random.txt 文件中。

```
import java.io.*;
import java.util.*;
```

```
class PrintStreamDemo{
     public static void main(String[] aa) throws Exception{
          PrintStream out = new PrintStream(new FileOutputStream("random.
txt"));
          Random r = new Random();  //生成一个 random 类对象 r
          int a;
          for(int i=0;i<10;i++){
               a = r.nextInt(100); //生成 0～99 之间的一个随机数并存入变量 a 中
               out.print(a+"\t");
          }
          out.close();              //关闭流
     }
}
```

在这个程序示例中，使用 java.util.Random 类生成 0～99 之间的随机数，并使用打印输出流 PrintStream 对象将它们写入 random.txt 文本文件中。

2．PrintWriter

向标准输出设备的输出流，可直接用来输出各种类型的数据。对 BufferedReader 类，该类的 readLine()方法能一次从流中读入一行，但对于 BufferedWriter 类，就没有一次写一行的方法，所以若要向流中一次写一行，可用 PrintWriter 类将原来的流改造成新的打印流，PrintWriter 类有一个方法 println()，能一次输出一行。例如：

```
PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter("D:\
javacode\test.txt")));
out.println("Hello World!");
out.close();
```

向文本输出流打印对象的格式化表示形式。此类实现在 PrintStream 中的所有 print 方法。此类中的方法不会抛出 I/O 异常。

20.5 实现用户输入

Java 中大多是通过定义输入 / 输出流对象来实现数据的输入和输出的，但是有时也需要通过键盘获得用户的输入。Java 提供了 java.util.Scanner 类，可以直接接收控制台命令行的输入。下面就介绍三种实现用户输入的常用方法。

20.5.1 使用 System.in 获取用户输入

Java 遵循标准 I/O 的模型，提供了 Syetem.in、System.out 及 System.err 类。本书一直都在用 System.out 往标准输出上写，而它(System.out)是一个已经预先处理过的、被包装成 PrintStream 的对象。和 System.out 一样，System.err 也是一个 PrintStream，但是 System.in 就不是了，它是一个未经处理的 InputStream。也就是说，虽然可以直接往 System.out 和 System.err 上写，但是要想读 System.in 的话，就必须先做处理了。

【例 20.12】写一个程序，使用 System.in 获取用户从键盘上的输入。

```
import java.io.*;
public class ReadKeyboardDemo{
     public static void main(String args[]) throws IOException{
```

```
        int a;
        System.out.print("请输入一个字符：");
        a = (char)System.in.read();        //获取键盘输入并存入变量 a 中
        System.out.println("你输入的字符是："+a);
    }
}
```



read()方法被定义为抽象方法，主要是为了让继承 InputStream 类的子类可以针对不同的外部设备定义不同的 read()方法。另外 Java 规定 read()必须配合异常处理机制来使用。

20.5.2　使用 Scanner 类获取用户输入

java.util.Scanner 类是 JDK 新增的一个类，可是使用该类创建一个从命令行读取数据的对象，而不必再进行流的转换。Scanner 类的使用方法如下所示。

```
Scanner reader=new Scanner(System.in);
```

然后 reader 对象调用下列方法，读取用户在命令行输入的各种数据类型：next.Byte()、nextDouble()、nextFloat()、nextInt()、nextLine()、nextLong()、nextShort()。这些方法执行时都要等待用户在命令行输入数据按回车确认。

【例 20.13】写一个程序，使用 Scanner 类获取用户输入，并计算输入值的和。

```
import java.util.*;
public class ReadKeyboardDemo2{
    public static void main(String args[]){
        System.out.println("请输入若干个数，每输入一个数用回车确认");
        System.out.println("最后输入一个非数字结束输入操作");
        Scanner reader=new Scanner(System.in);
                                        //创建读取命令行内容的 Scanner 对象
        double sum=0;
        int m=0;
        while(reader.hasNextDouble()){          //如果持续读入数据
            double x=reader.nextDouble();       //将读入的字符串转换为小数
            m=m+1;
            sum=sum+x;
        }
        System.out.println(m+"个数的和为:"+sum);
        System.out.println(m+"个数的平均值是:"+sum/m);
    }
}
```

在这个程序中，创建了一个 Scanner 类的对象 reader，用来读取命令行输入的内容。通过 reader 对象的 hasNextDouble()方法，来判断是否还有后续的 double 类型的输入内容。如果有的话，通过其 nextDouble()方法，将读入的字符串转换为小数，并累加到变量 sum 上。程序最后，输出 sum 的值和平均值。

20.5.3　使用命令行参数

在前面介绍的 Java 程序程序中，main()方法的参数从来没有被应用。实际上，main()

方法的 String 类型数组用来接收用户在执行应用程序时输入的参数。在运行 Java 应用程序的字节码文件时，用户可输入多个参数。其格式为：

```
Java    应用程序名   参数列表
```

参数之间及参数与应用程序名称之间要用空格隔开，按【Enter】键执行程序后，这些参数由 main()方法的参数 args[]接收，并在程序的语句中使用。

【例 20.14】计算 main()方法参数的和。

```
public class ReadKeyboardDemo3{
     public static void main(String args[]){
          int n=args.length;          //获取命令行参数的长度
          int sum=0;
          if(n==0)                    //如果命令行参数数组长度为 0
               System.out.println("没有参数! ");
          else{
               for(int i=0;i<n;i++)
                    sum=sum+Integer.parseInt(args[i]);
                                       //将命令行参数转换为整型并加到 sum 变量上
               System.out.println("参数的和为"+sum);
        }
    }
}
```

程序运行的效果如图 20.5 所示。

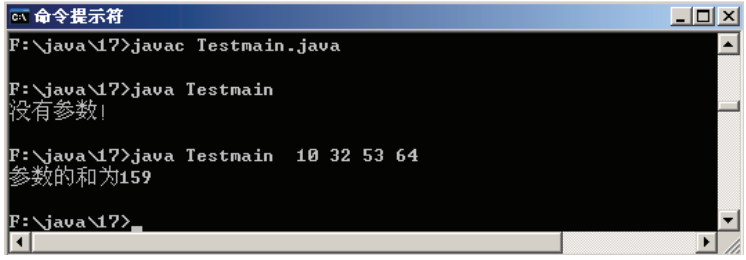


图 20.5　例 20.14 的运行结果

在这个程序中，运行时提供的命令行参数会自动地保存在数组 args 中，在程序中使用 for 循环来遍历 args 中的参数。因为从命令行中输入的参数内容都是以字符串类型存放的，所以要先使用 Integer 包装类的 parseInt 方法将其转换为整数，然后再相加。

20.6　小结

本章主要介绍了 Java 中的数据流，包括输入流和输出流。流的处理是 Java 中很重要的一项技术。读者在学习本章内容时，重点掌握对字节流的处理和对字符流的处理。Java 中流的体系比较复杂，可能会对读者的理解带来困难。建议读者在学习时重点放在字节流、字符流和缓冲流的处理上，其他的部分作为了解内容即可。下一章会介绍多线程的概念，讲解 Java 并发程序的设计。

第21章 并发程序设计

Java 平台被设计为从根本上就支持并发程序，在 Java 程序设计语言和 Java 类库中有基本的并发支持。此外，Java 平台还包括高层次的并发 API，这些高层次的并发 API 都包含在 java.util.concurrent 包中。本章就重点讲述并发程序设计的一些基本概念及 Java 的并发程序设计。



21.1 进程和线程

在并发程序中，有两个基本的执行单位：进程和线程。在 Java 程序设计语言中，并发程序主要集中于线程。然而，进程也很重要。

计算机系统内通常有许多动态的进程和线程。在单核的系统中，在任意时刻只能有一个线程在执行。单核系统中的处理时间通过操作系统的名为“时间片”的特性而被进程和线程所共享。

随着越来越多的计算机系统拥有多个处理器或带有多个执行内核（execution cores），系统并发执行进程和线程的系统能力也得到了极大的增强（但是对于不带多个处理器或执行内核的单系统来说，并发也是可能的）。

21.1.1 进程

程序是一段静态的代码，当一个程序启动时，它可以为即将开始的每项任务创建一个进程。进程是程序的一次动态执行过程，它对应了从代码加载、执行至执行完毕的一个完整过程，这个过程也是进程本身从产生、发展至消亡的过程。

一个进程（process）有完全独立的执行环境。一个进程通常有完全的、私有的一套基本的运行时资源，每个进程有它自己的内存空间。

进程经常看上去与程序或应用程序同义。然而，用户所看到的所谓“单一的应用程序”实际上可能是一系列协同运转的进程。要使进程之间的通信能顺利进行，大多数操作系统支持“内部进程通信（IPC）”（Inter Process Communication）资源，例如管道和套接字（Socket）。IPC 不仅用于同一系统上的进程间通信，而且用于不同系统上的进程间通信。

Java 虚拟机的大多数实现是作为一个单进程运行的。一个 Java 应用程序可以使用 ProcessBuilder 对象创建额外的进程。多进程的应用程序超出了本书的范围。

21.1.2 线程

线程与进程相似，是一段完成某个特定功能的代码，是程序中单个顺序的流控制。但与进程不同的是，同类的多个线程共享一块内存空间和一组系统资源，而线程本身的数据通常只有微处理器的寄存器数据，以及一个供程序执行时使用的堆栈。所以系统在产生一个线程，或者在各个线程之间切换时，负担要比进程小得多，因此，线程有时又称为轻量级进程（lightweight processes）。进程和线程都提供一个执行环境，但是创建一个新的线程要求的资源要比创建一个新的进程所要求的资源少得多。

线程存在于一个进程之中，每一个进程至少有一个线程。一个进程中可以包含多个线程。线程共享进程的资源，包括内存和打开的文件。

多线程执行是 Java 平台的一个本质特性。每一个应用程序有至少一个线程（或者如果计算系统线程的话，如内存管理和信号处理，则是多个）。但是从应用程序程序员的角度，启动的只有一个线程，称为主线程（main thread）。这个线程有能力创建另外的线程。

21.2 线程对象

每个线程都与 Thread 类的一个实例相关联。使用 Thread 对象来创建一个并发应用程序有两种基本类型：

- 直接控制线程的创建和管理，每当应用程序需要开始一个异步任务时，简单地实例化 Thread 类。
- 从应用程序的其他部分抽象出线程管理，传递应用程序的任务给一个执行器（executor）。

21.2.1 定义并启动一个线程

创建一个 Thread 类的实例的应用程序，必须提供将要在那个线程中运行的代码。有两种途径可以实现：提供一个 Runnable 对象和使用 Thread 的子类。

1．提供一个 Runnable 对象

Runnable 接口定义了一个单一的方法 run()，在 run()方法中包含要在线程中执行的代码。Runnable 对象被传递给 Thread 的构造器中，如下面的 RunnableDemo.java 所示。

【例 21.1】使用 Runnable 接口创建线程。

```
public class RunnableDemo implements Runnable {
    public void run() {     //实现 Runnable 接口，则必须实现该接口中的 run 方法
        System.out.println("这是一个线程!");
    }
    public static void main(String[] args) {
        (new Thread(new RunnableDemo())).start();
    }
}
```

2．Thread 的子类

Thread 类已经实现了 Runable 接口，不过其 run()方法什么也不做。应用程序可以通过



子类化 Thread，实现其自身的 run()方法。如下面的 ThreadDemo.java 所示。

【例 21.2】使用 Thread 的子类创建线程。

```
public class ThreadDemo extends Thread {
    public void run() {                          //重写从 Thread 继承过来的 run 方法
        System.out.println("这是一个线程!");
    }
    public static void main(String[] args) {
        (new Thread(new ThreadDemo())).start();
    }
}
```

请注意，两个示例程序都通过调用 Thread.start()方法来启动新的线程。两种用法应该用哪一种呢？对于第一种用法，使用了一个 Runnable 对象，是比较常用的一种，因为 Runnable 对象可以子类化一个非 Thread 类。第二种方法在简单的应用程序中更容易使用，但是它限制了完成异步任务的类只能从 Thread 类继承。本节主要使用第一种方法，它可以将 Runnable 的任务与执行此任务的线程对象（Thread 对象）分开。这种方式不仅更加灵活，而且也是稍后要讲到的高级线程管理 API 所用的方式。

Thread 类定义了许多用于线程管理的方法。这些方法包括静态方法，用来提供调用此方法的线程的信息，或影响调用此方法的线程的状态。还有其他的方法，可以从其他的线程调用，用于管理线程和 Thread 对象。

21.2.2　使用 Sleep 暂停线程执行

Thread.sleep()方法可以引起当前线程挂起执行一个指定的时期。这意味着处理时间可用于程序其他线程，或者运行在计算机系统内的其他应用程序。sleep()方法还被用于控制步调、等待另外的线程完成有时间要求的任务。

有两个重载的 sleep()方法。一个指定休眠的时间为毫秒，另一个指定休眠的时间为微秒（十亿分之一秒）。不过，这些休眠时间并不保证精确，这由后台的操作系统所提供的工具所限制。另外，休眠期也可以被中断信号所终结。

在任何情况下，都不能假定对 sleep()方法的调用会精确地按指定的时间周期挂起线程。下面的 SleepMessagesDemo.java 示例使用 sleep()方法每 3 秒输出一个消息。

【例 21.3】使用 Thread.sleep 方法，每隔 3 秒打印输出一个消息。

```
public class SleepMessagesDemo{
    public static void main(String[] args) throws InterruptedException {
        String messages[] = {
            "消息 1",
            "消息 2",
            "消息 3",
            "消息 4"
        };
        for (int i = 0; i < messages.length; i++) {
            Thread.sleep(3000);                  //执行到此暂停 3 秒
            System.out.println(messages[i]);     //输出一个消息
        }
    }
}
```

在上面的程序代码中，main()方法声明其抛出 InterruptedException 异常。当另一个线程中断了当前线程的 sleep 状态时，sleep()方法会抛出此异常。因为这个应用程序并没有定义另外的线程，所以不必担心会引起中断和捕获 InterruptedException 异常。

21.2.3 中断线程

所谓中断，是指示一个线程，它应该停止当前正在做的事情，转而去做一些其他事情。这由程序员来决定一个线程如何响应一个中断。不过更经常发生的是结束线程。

一个线程通过调用 Thread 对象的 interrupt()方法发出一个中断信号给要中断的线程。要使中断机制正确地工作，被中断的线程必须支持自身的中断。线程如何支持其自身的中断呢？

如果线程频繁调用会抛出 InterruptedException 异常的方法，那么在它捕获异常之后，简单地从 run()方法返回。例如，在上面的 SleepMessagesDemo.java 中，是在 Runnable 接口的 run()方法中循环输出消息。可以将其修改成如下所示代码以支持中断。

```
for (int i = 0; i < messages.length; i++) {
    try{
        Thread.sleep(3000);                  //执行到此暂停 3 秒
    }catch(InterruptedException e){
        return;                              //没有更多的消息了，中断线程
    }
    System.out.println(messages[i]);             //输出一个消息
}
```

许多抛出 InterruptedException 的方法（如 sleep()），被设计用于取消它们当前的操作，并且当中断信号到达时，立即返回。

如果一个线程运行很长时间而没有调用，会抛出 InterruptedException 异常的方法会怎么样呢？它必须周期性地调用 Thread.interrupted 方法来判断是否有中断信号到达。当一个中断信号到达时，返回 true。例如：

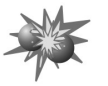
```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i];
    if(Thread.interrupted()){
        return;
    }
}
```

在这个简单的例子中，代码简单地判断中断，当中断信号到达时，退出线程。在更复杂的应用程序中，更倾向于抛出 InterruptedException()。例如：

```
if(Thread.interrupted()){
    throw new InterruptedException();
}
```

这样可以将中断处理代码集中在一个 catch 语句中。中断机制是通过使用称为“中断状态”的内部标记实现的。调用 Thread.interrupt 方法会设置这个标记。当一个线程通过调用静态方法 Thread.interrupted 检查中断时，中断状态被清除。非静态的方法 Thread.isInterrupted 被一个线程使用来查询另一个线程的中断状态，而不改变中断状态标记。

习惯上，任何通过抛出 InterruptedException 异常退出的方法，都会清除中断状态。然而，总是很有可能中断状态会立即被再次设置，被另外的线程调用 interrupt()方法。



21.2.4　join 方法

线程对象的 join()方法允许一个线程等待另外一个线程的完成。例如，一个当前正在执行的线程 t，它是 Thread 对象，如果调用其 join()方法，如下所示。

```
t.join();
```

这时会引起当前的线程暂停执行，直到 t 的线程终结为止。重载 join()方法允许程序员来指定一个等待的周期。不过与 sleep()一样，join 指定的这个时间也依赖于操作系统，所以不应该假设 join()方法会精确地等待指定的时间。

与 sleep()方法一样，join()方法响应中断时，会抛出一个 InterruptedException 并退出。例如，首先看下面这段代码，并判断其输出结果：

```
public class ThreadTest implements Runnable {
    public static int a = 0;            //定义静态变量
    public void run() {                 //在 run 方法里，循环增加静态变量 a 的值
        for (int k = 0; k < 5; k++) {
            a = a + 1;
        }
    }
    public static void main(String[] args) throws Exception {
        Runnable r = new ThreadTest();  //创建 ThreadTest 对象
        Thread t = new Thread(r);       //创建线程 t
        t.start();                      //启动线程 t
        System.out.println(a);          //输出静态变量 a 的值
    }
}
```

编译并运行上述程序，输出结果是 5 吗？答案是：有可能。其实很难遇到输出 5 的时候，通常情况下都不是 5（当然这也和机器有密切的关系）。因为当主线程 main()方法执行“System.out.println(a);”这条语句时，线程还没有真正开始运行（或许正在为它分配资源准备运行吧）。因为为线程分配资源需要时间，而 main()方法执行完 t.start()方法后继续往下执行“System.out.println(a);”，这个时候得到的结果是 a 还没有被改变的值 0。

如果想让输出结果为 5，就需要调用 join()方法。当调用 join()方法时，当前的 main 线程会阻塞执行，直到调用该方法的线程执行完毕 main 线程才会继续执行。下面是使用 join()方法以后的代码示例：

```
public class ThreadTest implements Runnable {
    public static int a = 0;            //声明静态变量 a
    public void run() {
        for (int k = 0; k < 5; k++) {
            a = a + 1;
        }
    }
    public static void main(String[] args) throws Exception {
        Runnable r = new ThreadTest();
        Thread t = new Thread(r);       //创建另一个线程 t
        t.start();                      //启动线程 t
        t.join();                       //当前线程阻塞执行，等待线程 t 执行完毕
        System.out.println(a);
    }
}
```

在调用线程 t 的 start()方法启动线程 t 以后，先调用 t.join()方法，等待线程 t 结束，main 线程才继续执行后续的“System.out.println(a);”方法，输出 a 的值，此时一定会输出结果 5。

21.2.5 实例：SimpleThreadsDemo.java

下面的程序实例 SimpleThreadsDemo.java 应用了本节介绍的线程的几个概念。在 SimpleThreadsDemo 中有两个线程。第一个线程是每个 Java 应用程序都有的 main 主线程。在主线程中创建一个新的实现了 Runnable 接口的 MessageLoop 线程，并等待它结束。如果 MessageLoop 线程的时间太长，主线程会中断它。

MessageLoop 线程输出一系列消息。如果在它输出所有的消息之前被中断，则 MessageLoop 线程输出信息并退出。

【例 21.4】线程使用实例。程序完整代码如下所示。

```
package chapter21;
public class SimpleThreadsDemo{
    //显示消息，消息前是当前线程的名字
    static void printThreadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message); //格式化输出线程信息
    }
    //私有静态内部类，实现了 Runnable 接口
    private static class MessageLoop implements Runnable {
        public void run() {
            String info[] = {"消息 1","消息 2","消息 3","消息 4"};
            try {
                for (int i = 0; i < info.length; i++) {
                    Thread.sleep(4000);                          //暂停 4 秒
                    printThreadMessage(info[i]);                 //输出消息
                }
            } catch (InterruptedException e) {
                printThreadMessage("不能正常工作。");
            }
        }
    }
    public static void main(String args[]) throws InterruptedException {
        //在中断 MessageLoop 线程之前延迟的毫秒数（默认是一个小时）
        long delay = 1000 * 60 * 60;
        //如果有命令行参数，那么在命令行参数中给出延迟的时间
        if (args.length > 0) {
            try {
                delay = Long.parseLong(args[0]) * 1000;
                                          //将命令行中的第一个参数解析为整数（秒）
            } catch (NumberFormatException e) {
                System.err.println("参数必须是整数.");
                System.exit(1);
            }
        }
        printThreadMessage("启动 MessageLoop 线程......");
        long startTime = System.currentTimeMillis();    //获得当前系统时间
        Thread t = new Thread(new MessageLoop());       //创建线程 t
```



```
            t.start();                          //启动线程 t
            printThreadMessage("等待 MessageLoop 线程结束......");
            //循环直到 MessageLoop 线程退出
            while (t.isAlive()) {
                printThreadMessage("继续等待...");
                //最多等待 1 秒钟等待 MessageLoop 线程结束
                t.join(1000);                   //main 线程暂停执行 1 秒，等待线程 t 结束
                //如果线程 t 运行的时间超过了 delay 指定的时间
                if (((System.currentTimeMillis() - startTime) > delay) && t.
isAlive()) {
                    printThreadMessage("时间太久了，不再等待!");
                    t.interrupt();              //中断线程 t
                    t.join();                   // main 线程暂停执行，直到线程 t 结束为止
                }
            }
            printThreadMessage("MessageLoop 线程结束!");
        }
    }
```

代码说明：

（1）在 SimpleThreadsDemo 类中，首先定义了一个方法 printThreadMessage，如下所示。

```
    static void printThreadMessage(String message) {
           String threadName = Thread.currentThread().getName();
           System.out.format("%s: %s%n", threadName, message); //格式化输出线程信息
    }
```

方法 printThreadMessage()方法用来格式化输出线程的信息。其中要输出的消息以参数的形式传入。在方法内，通过调用 Thread 类的 currentThread()方法，获得对当前正在执行的线程对象的引用，然后调用当前线程对象的 getName()方法，获得当前线程的名字。方法的最后一行，使用 System.out.format()方法，格式化输出参数信息。其中%s 代表此处输出的是一个字符串变量，实际输出值为后续相对应的参数值。%n 代表输出以后换行。在程序的任何位置都可以调用此方法显示当前线程的信息。

（2）接着创建一个实现 Runnable 接口的类 MessageLoop，并实现 Runnable 接口中定义的方法 run()，在 run()方法中定义线程每输出一个消息字符串，就休眠 4 秒钟。在休眠期间，有可能会发生 InterruptedException 异常，因此要予以捕获。

（3）最后，定义主线程方法 main()。在 main()方法中，通过将 Runnable 对象传递给 Thread 的构造器中，创建一个线程对象 t，并调用 t.start()方法启动线程 t。然后使用 while 循环，在 while 循环中对线程 t 进行判断，通过调用 t.isAlive()方法判断线程 t 是否仍然处于活动状态。如果仍然处于活动状态，就调用线程对象 t 的 join 方法，并传入参数 1000，表示当前线程暂停执行 1 秒的时间，等待线程 t 的运行结束。如果等待 1 秒以后，再次判断发现线程 t 仍然处于活动状态，并且总等待时间超过了 delay 指定的时间，那么就在主线程 main 中调用线程 t 的 interrupt()方法，中断线程 t 的执行。

需要注意的是，调用线程 t 的 interrupt()方法并不能保证线程 t 就会立即中断。所以最后调用 t 的不带参数的 join()方法，停止当前主线程的执行，直到 t 运行结束为止。

（4）总延迟时间，需要在运行此程序时，在命令行参数中传入，然后使用 Long.parseLong(args[0])方法将其解析为整数，并转换为毫秒。如果不从命令行中指定的话，

默认为延迟一个小时，即 main()方法中第一行代码所表示的：

```
long delay = 1000 * 60 * 60;
```

最后，使用如下命令编译此程序，并使用不带命令行参数的 java 命令运行此程序：

```
javac -d . SimpleThreadsDemo.java
java chapter21.SimpleThreadsDemo
```

运行和输出结果如下所示。

```
main: 启动 MessageLoop 线程......
main: 等待 MessageLoop 线程结束......
main: 继续等待...
main: 继续等待...
main: 继续等待...
main: 继续等待...
Thread-0: 消息 1
main: 继续等待...
main: 继续等待...
main: 继续等待...
main: 继续等待...
Thread-0: 消息 2
main: 继续等待...
main: 继续等待...
main: 继续等待...
main: 继续等待...
Thread-0: 消息 3
main: 继续等待...
main: 继续等待...
main: 继续等待...
main: 继续等待...
main: 继续等待...
Thread-0: 消息 4
main: MessageLoop 线程结束!
```

如上面的输出结果所示，每个线程都有一个标识名，多个线程可以同名。如果线程创建时没有指定标识名，就会为其生成一个新名称。

使用带命令行参数的 java 命令运行此程序：

```
java chapter21.SimpleThreadsDemo 3
```

程序运行和输出结果如下所示。

```
main: 启动 MessageLoop 线程......
main: 等待 MessageLoop 线程结束......
main: 继续等待...
main: 继续等待...
main: 继续等待...
main: 时间太久了，不再等待!
Thread-0: 不能正常工作。
main: MessageLoop 线程结束!
```

可以看出，由于线程 t 运行的时间超过了 delay 指定的时间，所以线程 t 被调用 interrupt()方法进行中断。



21.3　同步

线程间的通信主要是通过共享访问字段和对象引用实现的。这种形式的通信非常有效，但是有可能会造成两种类型的错误：线程冲突（thread interference）和内存一致性错误（memory consistency errors）。要防止这些错误，就需要使用同步。

21.3.1　线程冲突

在理解线程冲突的含义之前，先来看一个简单的类 Counter 的定义：

```
class Counter {
   private int c = 0;

   public void increment() {
      c++;
   }

   public void decrement() {
      c--;
   }

   public int value() {
      return c;
   }

}
```

在类 Counter 中，每当调用 increment()方法时，变量 c 的值就加 1，每当调用 decrement()方法时，变量 c 的值就会减 1。这在单一的线程中没有问题，但是当从多个不同的线程中引用 Counter 对象并调用 increment()方法和 decrement()方法时，变量 c 的值可能就不是这样改变的。

当在不同的线程中执行两个操作进行交叉存取，但操作对象是同一数据时，就会发生冲突。例如，在单独的线程中，对表达式 c++的操作可以分解为三步：

（1）获得变量 c 当前的值。

（2）将获得的变量 c 的值加 1。

（3）将改变（增加）以后的值再赋给变量 c。

表达式 c--的操作也可以分解为这三步。

现在假设一个线程 A 调用方法 increment()，同时另一个线程 B 也在同一时刻调用方法 decrement()。如果变量 c 的初始值为 0，那么就可能会发生以下的交叉操作：

（1）线程 A 获得变量 c 的当前值（为 0）。

（2）线程 B 获得变量 c 的当前值（为 0）。

（3）线程 A 将获得的值增加 1，增加结果为 1。

（4）线程 B 将获得的值减少 1，减少结果为-1。

（5）线程 A 将增加后的值保存回变量 c，此时 c 的值改变为 1。

（6）线程 B 将减少后的值保存回变量 c，此时 c 的值改变为-1。

操作的结果是，线程 A 的值最后丢失了，被线程 B 的值所覆盖。在实际执行过程中，

也可能是线程 B 的值丢失而最后保留线程 A 的值，或者根本就没有错误。因为不可预期，所以线程冲突这种 bug 很难被检测和修正。

上面的问题所反映的实质是，在 Java 的多线程程序中，当两个或多个线程同时访问同一个变量，并且一个线程需要修改这个变量时，应对这样的问题做出处理，否则可能发生混乱。

21.3.2 内存一致性错误

当不同的线程对同一数据获得不同的值时，就会发生内存一致性错误。既两个线程对同一内存的读写相互之间是不可见的。例如，假设定义了一个简单的 int 变量并初始化如下。

```
int counter = 0;
```

假设两个不同的线程 A 和 B 共享变量 counter。如果线程 A 执行改变变量 counter 的值的操作，代码如下所示。

```
counter++;
```

而同时线程 B 执行输出变量 counter 值的操作，代码如下所示。

```
System.out.println(counter);
```

那么输出结果可能为 1，但是也有可能为 0，因为线程 B 在输出变量 counter 的值的时候，可能并不知道线程 A 改变了变量 counter 的值。

要避免内存不一致性错误，关键是在线程之间建立一种 happens-before 关系，这种关系可以保证一个语句对内存的写操作可以被另外一个指定的语句所知道。要创建这种 happens-before 关系，可以通过同步来实现。在以下两种情况下，会自动创建 happens-before 关系：

- 当一个语句调用 Thread.start()时，每一个与此语句有 happens-before 关系的语句，都会与新创建的线程所执行的每一个语句建立 happens-before 关系。因此导致新的线程创建的代码所发生的操作对新的线程也是可见的。
- 当一个线程停止并调用另一个线程的 Thread.join()时，所有停止线程中执行的语句与 join()成功后的所有语句都有一个 happens-before 关系。因此线程中的操作的结果对执行 join()以后的线程都是可见的。

21.3.3 同步方法

Java 语言提供两个基本的同步用法：同步方法和同步语句。其中同步语句相对比较复杂，将在下一节讨论。本节主要介绍同步方法。

把修改数据的方法用关键字 synchronized 来修饰。一个方法使用关键字 synchronized 修饰后，当一个线程 A 使用这个方法时，其他线程想使用这个方法时就必须等待，直到线程 A 使用完该方法。

要创建同步方法，只需要简单地在方法声明前加上关键字 synchronized 即可。例如，将上一节中的 Counter 类中的方法改写为同步方法，如以下代码所示。

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
```



```
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

当方法被声明为同步后，可以保证在同一对象上对同步方法的两次交错调用这种情况不可能发生。当一个线程执行对象上的同步方法时，所有其他调用同一对象同步方法的线程会阻塞（挂起执行），直到第一个线程完成对该对象同步方法的调用。

当同步方法退出时，它会自动地与任何对同一对象同步方法的后续调用建立 happens-before 关系。这保证对象状态的改变对所有的线程都是可见的。

另外，不能将构造器方法声明为同步的，这没有意义，而且是一个语法错误。同步方法为防止线程冲突和内存一致性错误提供了简单而有用的策略：如果一个对象可以被多个线程使用，那么对该对象的变量的所有读写都要通过同步方法来实现。只有一个例外：当变量被定义为 final 时，这时是不能修改 final 类型的变量的，所以可以通过非同步方法安全地访问这一类型的变量。

21.3.4　固定锁和同步

同步是构建在一个称为“固定锁”或“监视锁”的内部实体上的。固定锁在同步的各个方面都扮演着重要的角色：增强对一个对象状态的排他性访问（互斥）及建立 happens-before 关系。

每一个对象都有一个相关联的固定锁。按惯例，一个需要对一个对象的字段进行排他性和持续性访问的线程，在访问之前要获得对象的固定锁，然后在访问以后释放固定锁。在获得和释放固定锁期间称一个线程拥有固定锁。只要一个线程拥有一个固定锁，其他线程就不能获得同一固定锁。如果其他线程试图获得锁，将会阻塞。

当一个线程释放一个固定锁时，就会在此动作和任何后续的获得同一锁的动作之间建立 happens-before 关系。

当一个线程调用一个同步方法时，它自动地获得该方法所属对象的固定锁，并且当方法返回时，线程会释放这个锁。即使方法的返回是由一个未捕获的异常引起的，该固定锁也会被释放。

当一个静态同步方法被调用时，因为静态方法是与类而不是与对象相关联的，因此线程会获得与该类所关联的 Class 对象的固定锁。所以控制对类的静态字段的访问的锁与用于该类的任何实例对象的锁不同。

除了同步方法之外，另外一个创建同步代码的方式是同步语句。与同步方法不同，同步语句必须指定提供固定锁的对象。例如下面的代码：

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

在上面这段程序代码中，addName()方法需要同步地改变 lastName 和 nameCount，但也需要避免其他对象的方法的同步调用。如果没有同步语句，为了达到与调用 nameList.add() 相同的目的，将不得不设计很多分开的、非同步的方法。

同步语句还有助于提高细粒度的同步并发。例如，假设一个类 Lunch 有两个实例字段，m1 和 m2，这两个字段永远不会一起使用。所有对于这些字段的更新必须被同步，但是没有理由禁止对 m1 和 m2 的交错更新，否则将会创建不必要的阻塞，从而降低并发性。代替使用同步方法或其他使用与 this 相关联的锁，这里创建两个单一的对象来提供锁。

```
public classLunch {
    private long m1 = 0;
    private long m2 = 0;
    private Object lock1 = new Object();   //创建对象 lock1 作为一个锁
    private Object lock2 = new Object();   //创建对象 lock2 作为一个锁
    public void addm1() {                  //更新 m1 的方法
        synchronized(lock1) {              //使用同步锁 lock1，同步对 m1 的更新
            m1++;
        }
    }
    public void addm2() {                  //更新 m2 的方法
        synchronized(lock2) {              //使用同步锁 lock2，同步对 m2 的更新
            m2++;
        }
    }
}
```

一个线程不能获得属于另外一个线程的锁。但是一个线程可以获得它已经拥有的锁。允许一个线程获得同一把锁多次可以再次进入同步。如果存在同步的代码，在同步代码中，直接或间接地调用也包含有同步代码的方法，并且两批代码使用同一个锁。没有再次进入同步，同步代码将不得不采取许多额外的预防措施来避免一个线程引起它自身的阻塞。

21.3.5 原子访问

所谓原子操作，就是“不可中断的一个或一系列操作”。一个原子操作不能在中间停止：要么操作全部完成，要么根本就不发生。原子操作的作用只到操作全部完成才能显现。

在前面讲到过自增表达式，如 C++，可以分解为三步更加小的操作过程，因此它不是一个原子操作。甚至于一些非常简单的表达式也可以定义很复杂的操作，可以被分解为其他操作。不过，程序员可以指定其为原子操作：

- 存取和更新除了 long 和 double 之外的任何类型的成员变量所对应的存储单元，它们都是原子的。其中包括引用其他对象的成员变量。
- 对于所有声明为 volatile 类型的变量（包括 long 和 double 类型，即 volatile long 和 volatile double 类型）来说，存取都是原子的。

原子操作不能交叉存取，所以可以放心使用，而不必担心线程冲突的问题。然而，这并不能消除所有对同步原子操作的需要，因为内存一致性错误仍然可能存在。使用 volatile 类型的变量可以减少内存一致性错误的风险，因为任何对一个 volatile 变量的写操作都是建立在一个与后续的对同一变量的读操作的 happens-before 关系之上的。这意味着对



volatile 变量的改变对其他线程来说总是可见的。

更进一步，这还意味着当一个线程读取一个 volatile 变量时，它看到的不仅仅是对 volatile 的最后的改变，而且还包括导致这种改变的代码的影响。

使用简单的原子变量访问比通过同步代码访问这些变量更有效率，但要求程序员要更加小心以避免内存一致性错误。这样额外的努力是否必要，取决于应用程序的大小和复杂性。在 java.util.concurrent 包中的一些类提供了不依赖于同步的原子方法。



21.4　活跃性

一个并发应用程序能以合适的行为去执行的能力，称为“活跃性（liveness）”。本节介绍最常见类型的存活问题，死锁（deadlock），并简单介绍两个其他的存活问题：饥饿（starvation）和活锁（livelock）。

21.4.1　死锁

死锁代表这样一种情况，两个或多个线程彼此等待对方因而永远地被阻塞。例如下面这种情况：张财主和李财主是一对好朋友，而且都非常有礼貌。作为非常有礼貌的一个表现是，当一个人向另一个朋友弯腰鞠躬致敬时，他必须保持弯腰鞠躬的姿势，直到他的朋友有机会回敬一个鞠躬为止。不过不幸的是，当两个好朋友同时向对方弯腰鞠躬致敬时，就会发生双方都等待对方鞠躬回敬的情形，双方不得不永远弯腰鞠躬下去。这就是死锁。在下面的死锁程序示例中，对这种可能性进行了模拟。

【例 21.5】死锁程序示例。在下面的程序中，定义了两个同步的方法 bow()和 bowBack()，分别代表向朋友鞠躬和朋友回敬鞠躬。当启动两个线程时，有可能会两个线程同时调用 bowBack()方法，这时就会发生死锁。

```
public class DeadlockDemo{
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {     //同步的鞠躬方法
            System.out.format("%s: %s 向我鞠躬致敬!%n",
                    this.name, bower.getName());
            bower.bowBack(this);                          //回敬鞠躬
        }
        public synchronized void bowBack(Friend bower) {//同步的回敬鞠躬方法
            System.out.format("%s: %s 对我鞠躬回敬!%n",
                    this.name, bower.getName());
        }
    }
    public static void main(String[] args) {
        final Friend zhang = new Friend("张财主");        //声明两个Friend对象
        final Friend li = new Friend("李财主");
```

```
        new Thread(new Runnable() {                    //创建并启动第 1 个线程
            public void run() { zhang.bow(li); }
        }).start();
        new Thread(new Runnable() {                    //创建并启动第 2 个线程
            public void run() { li.bow(zhang); }
        }).start();
    }
}
```

当程序 DeadlockDemo 运行时，很有可能当两个线程试图调用 bowBack()方法时，都会阻塞。两个阻塞都不会结束，因为每一个线程都在等待对方退出对 bow()的调用。

21.4.2 饥饿和活锁

饥饿（starvation）和活锁（livelock）发生的可能性比死锁要少得多，但是对于每一个并发软件设计者来说，仍然有可能会遇到。

饥饿描述了这样一种情形：一个线程不能获得对共享资源的经常性的访问，不能前进。这发生在由于“贪婪”线程对资源长期占用而导致资源长期不可用时。例如，假设一个对象提供一个同步方法，该同步方法经常需要很长的时间才能返回。如果一个线程频繁地调用这个方法，那么其他也需要频繁地同步访问同一对象的线程就会经常处于“饥饿”状态，因而经常阻塞。

一个线程会经常响应另一个线程的操作。如果另一个线程的操作也是为了响应第三个线程的操作，那么就会导致活锁的产生。与死锁一样，产生活锁的线程不能做进一步的处理。不过，该线程不会阻塞，它们只是因为太忙于彼此响应以至于不能继续工作。

活锁可以用下面的例子来比喻：张财主和李财主并排站立，张财主向其左边的李财主传递一个箱子，而同时李财主向其右边的张财主也传过来一个箱子。因为他们还是彼此阻塞，因此张财主转而向其右边的人传递，而同时李财主转而向其左边的人传递。这时，他们依然彼此阻塞。虽然传递工作不能继续，但是张财主和李财主两人一直在忙于不停地左右传递，他们并没有停止工作。

21.5 守护块

线程经常不得不调整它们的操作。最经常使用的调整惯例为“守护块”。守护块以循环检测一个条件开始，只有条件为 true 时，块才能继续进行下去。

例如，假设 guardedJoy()是一个方法，只有共享变量 joy 被另外的线程设置以后，guardedJoy()方法才能继续进行。从理论上来讲，这样的方法只需要进行简单的循环直到条件被满足为止，但是循环很浪费时间，因为在等待期间它会不断地执行。例如下面的代码：

```
public void guardedJoy() {
    //简单地循环守护。这很浪费时间，不要这样做
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

更有效的守护是调用 Object.wait()方法，挂起当前的线程。对 wait()的调用不会返回，除非另外的线程发布一个通知告知特定的事件已经发生了，即使发生的事件并不是本线程



必须等待的事件。如下面的代码所示。

```
public synchronized guardedJoy() {
    //这个守护为每一个指定的事件只循环一次，即使该事件并不是所等待的
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

像许多挂起的方法一样，wait 也能抛出 InterruptedException 异常。在上面的例子中，仅仅忽略了该异常，只关注 joy 的值。

为什么这个方法中要使用同步呢？假设 d 是一个对象，用它来调用 wait()方法。当一个线程调用 d.wait()时，它必须获得 d 的固定锁，否则就会抛出一个错误。在同步方法内调用 wait()方法是获得固定锁的一个简单的途径。

当 wait 方法被调用以后，线程会释放这个锁并挂起执行。在将来的某个时刻，另一个线程将获得同一把锁，并调用 Object.notifyAll()方法，通知正在等待该锁的所有线程。例如：

```
public synchronized notifyJoy(){
    joy = true;
    notifyAll();
}
```

在第二个线程释放该锁若干时间以后，第一个线程重新获得该锁，并从对 wait()的调用中返回，恢复执行。例如，在例 21.6 中，要传递的数据是一系列文本消息，通过类型为 Drop 的对象共享。

【例 21.6】“生产者-消费者”应用程序示例。

（1）首先创建由生产者和消费者所共享的对象：Drop 类。Drop 类中定义两个同步的方法，take()方法用来读取消息，而 put()方法用来发送消息。Drop 类位于 chapter21 包中，并单独保存为 java 源文件。

```
package chapter21;
public class Drop {
    //从生产者发往消费者的消息
    private String message;
    //如果消费者在等待生产者发送消息，为 true；如果生产者在等待消费者获取数据，为 false
    private boolean empty = true;
    public synchronized String take() {
        //消费者等待，直到有消息可用
        while (empty) {                          //循环等待条件 empty 成立
            try {
                wait();                          //等待其他线程唤醒
            } catch (InterruptedException e) {}
        }
        //改变状态
        empty = true;
        //通知生产者状态已经改变
        notifyAll();
        return message;
```

```
    }
    public synchronized void put(String message) {
        //等待,直到消息被取走
        while (!empty) {
            try {
                wait();                                  //等待其他线程唤醒
            } catch (InterruptedException e) {}
        }
        //改变状态
        empty = false;
        //存储消息
        this.message = message;
        //通知消费者状态已经改变
        notifyAll();
    }
}
```

（2）生产者线程定义在 Producer.java 中，它发送一系列消息。如果发送的消息为字符串“over”，说明所有的消息已经发送完毕。为了模拟真实世界应用程序的不可预知性，生产者线程会在发送消息中间暂停一个随机的时间。Producer 类仍然位于 chapter21 包中。

```
package chapter21;
import java.util.Random;
public class Producer implements Runnable {
    private Drop drop;                    //声明一个生产者和消费者所共享的对象
    public Producer(Drop drop) {          //构造器中初始化共享对象
        this.drop = drop;
    }
    public void run() {
        String importantInfo[] = {"消息 1","消息 2","消息 3","消息 4"};
        Random random = new Random();     //创建一个随机对象
        for (int i = 0; i < importantInfo.length; i++) {
                                          //循环向共享对象中放入消息
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt(5000));   //休眠一个随机时间
            } catch (InterruptedException e) {}
        }
        drop.put("over");                 //线程结束时，发出通知消息
    }
}
```

（3）消费者线程，定义在类 Consumer 中，简单地获取消息并打印输出。如果收到“over”字符串就结束。这个线程也暂停一个随机的间隔时间。同样，Consumer 类以单独的 java 文件存放，并位于包 chapter21 中。

```
package chapter21;
import java.util.Random;
public class Consumer implements Runnable {
    private Drop drop;                    //声明一个生产者和消费者所共享的对象
    public Consumer(Drop drop) {          //构造器中初始化共享对象
        this.drop = drop;
    }
    public void run() {
```



```
        Random random = new Random();    //创建一个随机对象
        String message = drop.take();    //从共享对象中获取生产者"生产"的消息
        while(! message.equals("over")){//如果不是结束的消息，则输出
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
            message = drop.take();       //继续从共享对象中获取"生产"的消息
        }
    }
}
```

（4）最后，在 ProducerConsumerDemo.java 文件中，定义主线程，用来启动生产者和消费者线程。

```
package chapter21;
public class ProducerConsumerDemo{
    public static void main(String[] args) {
        Drop drop = new Drop();              //创建生产者和消费者共享的对象
        (new Thread(new Producer(drop))).start();   //创建并启动第 1 个线程
        (new Thread(new Consumer(drop))).start();   //创建并启动第 2 个线程
    }
}
```

（5）使用如下的命令依次对上述文件进行编译。

```
javac -d . Drop.java
javac -d . Producer.java
javac -d . Consumer.java
javac -d . producerConsumerDemo.java
```

编译成功以后，在文件夹 chapter21 中就会看到这些类对应的.class 文件。然后用下面的命令运行此程序：

```
java chapter21.ProducerConsumerDemo
```

得到的运行结果如下所示。

```
获取的消息为：消息 1
获取的消息为：消息 2
获取的消息为：消息 3
获取的消息为：消息 4
```

上面的程序使用守护块来创建一个“生产-消费者”应用程序。这类应用程序在两个线程间共享数据：生产者创建数据，而消费者处理数据。这两个线程使用一个共享对象进行通信。一个基本的条件是，消费者线程一定不能在生产者线程分发数据前获得数据，同样，如果消费者线程还没有获取原来的数据，生产者线程一定不能分发新的数据。



21.6　不可变对象

如果一个对象在构建以后，其状态不能被改变，称此对象为“不可变的”。最大限度地依靠不可变对象是创建简单、可靠代码的一个策略。在并发程序中，不可改变的对象特别有用。因为它们不能改变状态，所以它们不能被线程冲突或不一致的视图所破坏。在下

面的部分内容中，介绍了一个类，其实例对象是易变的，以及从它派生出的另一个类，其实例对象是不可变的。通过这两个类，给出这一类转换的一般原则，并举例说明不可变对象的一些好处。

21.6.1 同步类实例

下面定义一个代表颜色的类 SynchronizedRGB。SynchronizedRGB 类的每一个对象都使用三原色的值和颜色的名称字符串来代表一个颜色。

【例 21.7】在包 chapter21 中创建同步类实例 SynchronizedRGB。

```
package chapter21;
public class SynchronizedRGB {
    // 声明组成颜色的 RGB 值，值必须在 0 和 255 之间
    private int red;
    private int green;
    private int blue;
    private String name;
    //对指定的参数值进行检查，看是否在正确的范围之内（0～255）
    private void check(int red, int green, int blue) {
        if (red < 0 || red > 255
                || green < 0 || green > 255
                || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }
    // 构造器中，先对传入的 RGB 颜色值进行检查，如果正确，则初始化字段的值
    public SynchronizedRGB(int red, int green, int blue, String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }
    //更改代表 RGB 颜色的属性字段的值
    public void set(int red, int green, int blue, String name) {
        check(red, green, blue);          //调用自定义方法对传入的参数进行检查
        synchronized (this) {             //同步块
            this.red = red;
            this.green = green;
            this.blue = blue;
            this.name = name;
        }
    }
    //同步方法获得 RGB 的值
    public synchronized int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }
    //同步方法获得颜色的值
    public synchronized String getName() {
        return name;
    }
    //同步方法改变 RGB 值
```



```
    public synchronized void invert() {
        red = 255 - red;
        green = 255 - green;
        blue = 255 - blue;
        name = "Inverse of " + name;
    }
}
```

例如，假设一个线程执行如下的代码：

```
SynchronizedRGB color = new SynchronizedRGB(0, 0, 0, "黑色");
...
int myColorInt = color.getRGB();                    //语句 1
String myColorName = color.getName();               //语句 2
```

这时，如果另外一个线程在语句 1 之后但在语句 2 之前调用 color.set()方法，那么 myColorInt 的值将与 myColorName 的值不匹配。要避免这种结果，语句 1 和语句 2 必须绑定在一起：

```
synchronized (color) {
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

这种类型的不一致只可能发生在可变对象上，如果 SynchronizedRGB 是不可变对象，则不会发生这种情况。

21.6.2　定义不可变对象的策略

下面定义了一个用于创建不可变对象的简单策略的规则。并不是所有标明为“不可变的”的类都遵守这样的规则。不过，这些策略是针对成熟的分析员的，不适合于初学者。这些规则包括：

- 不要提供 set 方法，这些方法用来修改字段或字段所引用的对象。
- 将所有的字段声明为 final 和 private。
- 不允许子类覆盖方法。最简单的办法是将类声明为 final 的。一个更加成熟的方式是将构造器声明为私有的并通过工厂方法构造实例。
- 如果实例的字段包括对一个易变对象的引用，不允许改变这些对象。这包括：不提供修改易变对象的方法；不共享指向易变对象的引用；永远不要将引用存储在外部，易变对象传递给构造器；如果必要的话，创建拷贝，并将引用存储在拷贝中。同样地，如果有必要避免在方法中返回原始对象，就创建内部易变对象的拷贝。

将这些规则应用到 SynchronizedRGB 上，做以下改变：

- 在 SynchronizedRGB 类中有两个 set()方法。第一个是 set，任何转换对象，在一个不可变的类中，应该取消掉。第二个是 invert()方法，将其调整为创建一个新的对象，而不是修改已经存在的对象。
- 所有的字段已经是 private 的了；进一步将其声明为 final 类型的。
- 将类本身声明为 final 类型的。
- 只有一个字段指向一个对象，而所指向的对象的本身是不可变的。

【例 21.8】应用以上规则，在包 chapter21 中创建不可变的对象。

```
package chapter21;
final public class ImmutableRGB {
    //值必须在 0～255
    final private int red;
    final private int green;
    final private int blue;
    final private String name;
    //对指定的参数值进行检查，看是否在正确的范围之内
    private void check(int red, int green, int blue) {
        if (red < 0 || red > 255
                || green < 0 || green > 255
                || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }
    // 构造器中，先对传入的 RGB 颜色值进行检查，如果正确，则初始化字段的值
    public ImmutableRGB(int red, int green, int blue, String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }
    // 非同步方法获得 RGB 的值
    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }
    // 非同步方法获得颜色的值
    public String getName() {
        return name;
    }
    // 非同步方法，创建一个新的颜色对象，并返回
    public ImmutableRGB invert() {
        return new ImmutableRGB(255 - red, 255 - green, 255 - blue,
                "Inverse of " + name);
    }
}
```

从上面的代码中可以看出，取消了容易改变对象状态的 set()方法，并且改写了 invert()方法，在 invert()方法中，创建并返回一个新的 ImmutableRGB 对象。

21.7 高级并发对象

到目前为止，介绍的都是一些低级的 API，它们很久以来就是 Java 平台的一部分。对于基本的任务来说，这些 API 足够了。但是对于更高级的任务，就需要高级构建块了。特别是在充分使用多处理器和多核系统的今天，重并发程序更需要高级并发处理。

大多数这些高级特性是通过新的 java.util.concurrent 包实现的。在 Java 集合框架中也有新的并发数据结构。这些新的高级并发对象包括：

- 锁对象（Lock objects），支持锁的习惯用法，简化许多并发应用程序。
- 执行器（Executors），定义一个启动和管理线程的高级 API。由 java.util.concurrent



提供的执行器提供与大规模应用程序相适应的线程池管理。

- 并发集合（Concurrent collections），并发集合使得管理大规模数据集合更加容易，并且能够极大地减少对于同步的需要。
- 原子变量（Atomic variables），具有将同步减小到最少和帮助避免内存一致性错误的特性。

21.7.1　锁对象

同步代码依赖于一个简单类型的可重复获得的锁。这种锁虽然很容易使用，但是有许多限制。java.util.concurrent.locks 包支持更加成熟的锁的用法。在 java.util.concurrent.locks 包中有个最基本的接口：Lock。

Lock 对象的工作方式与同步代码使用的隐式锁非常类似。使用隐式锁，一个时刻只有一个线程能拥有一个 Lock 对象。通过与其相关联的 Condition 对象，Lock 对象还支持 wait/notify 机制。

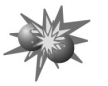
Lock 对象比隐式锁最大的好处是它们收回试图获取锁的努力的能力。如果锁不可用，tryLock()方法能够立即收回或在指定的期限前收回获取锁的企图。如果另外一个线程在锁被获得之前发出一个中断信号，lockInterruptibly()方法能收回对锁的获取。

下面使用 Lock 对象来解决死锁问题。在这里，张财主和李财主已经被培训过，当一个朋友鞠躬致敬时，要注意观察。现在建立这种改进的模型，要求在进行鞠躬之前，Friend 对象必须获得用于参与双方的锁。例 21.9 是这种模型的代码实现。

【例 21.9】使用 Lock 对象解决死锁问题。

```
package chapter21;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;
public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock(); //创建一个可重入的互斥锁
        public Friend(String name) {                   //构造器
            this.name = name;
        }
        public String getName() {                      //获取 name 字段的值
            return this.name;
        }
        public boolean impendingBow(Friend bower) {    //改写的鞠躬致敬方法
            Boolean myLock = false;
            Boolean yourLock = false;
            try {
                myLock = lock.tryLock();
                                //在调用时锁未被另一个线程保持的情况下，才获取该锁
                yourLock = bower.lock.tryLock();
            } finally {
                if (! (myLock && yourLock)) {
                    if (myLock) {
                        lock.unlock();                 //释放锁
```

```
                }
                if (yourLock) {
                    bower.lock.unlock();
                }
            }
        }
        return myLock && yourLock;
    }
    public void bow(Friend bower) {              //鞠躬致敬
        if (impendingBow(bower)) {
            try {
                System.out.format("%s: %s 向我鞠躬致敬!%n",
                        this.name, bower.getName());
                bower.bowBack(this);
            } finally {
                lock.unlock();
                bower.lock.unlock();
            }
        } else {
            System.out.format("%s: %s 开始向我鞠躬致敬,但" +
                    "但他已经看见我正在向他鞠躬致敬.%n",
                    this.name, bower.getName());
        }
    }
    public void bowBack(Friend bower) {
        System.out.format("%s: %s 已经向我鞠躬回敬!%n",
                this.name, bower.getName());
    }
}
static class BowLoop implements Runnable {
    private Friend bower;
    private Friend bowee;
    public BowLoop(Friend bower, Friend bowee) { //构造器，初始化两个对象
        this.bower = bower;
        this.bowee = bowee;
    }
    public void run() {
        Random random = new Random();            //创建随机对象
        for (;;) {
            try {
                Thread.sleep(random.nextInt(10)); //线程休眠一个随机时间
            } catch (InterruptedException e) {}
            bowee.bow(bower);
        }
    }
}
public static void main(String[] args) {
    final Friend zhang = new Friend("张财主");
    final Friend li = new Friend("李财主");
    new Thread(new BowLoop(zhang, li)).start(); //创建并启动第 1 个线程
    new Thread(new BowLoop(li, zhang)).start(); //创建并启动第 2 个线程
}
}
```



编译并运行此程序，会发现张财主和李财主对这种新的安全的鞠躬模式如此着迷，以至于他们情不自禁地不停地鞠躬（无限循环）。

在上面的代码中，创建了一个可重入的互斥锁 ReentrantLock，它具有与使用 synchronized 方法和语句所访问的隐式监视器锁相同的一些基本行为和语义，但功能更强大。ReentrantLock 将由最近成功获得锁，并且还没有释放该锁的线程所拥有。当锁没有被另一个线程拥有时，调用 lock 的线程将成功获取该锁并返回。如果当前线程已经拥有该锁，此方法将立即返回。

21.7.2　执行器

在前面所有的示例中，通过 Runnable 对象定义的新的线程和通过 Thread 对象定义的线程本身所执行的任务有非常紧密的连接。这在小的应用程序中没有问题，但是在大规模的应用程序中，会有意识地将线程管理和创建与程序的其他部分分离开来。封装这些功能的对象就是“执行器（Executors）”。

1．执行器接口

在 java.util.concurrent 包中定义了三个执行器接口：

- Executor，一个简单的接口，支持启动新的线程。
- ExecutorService，是 Executor 的子接口，添加了辅助管理生命周期的特性，包括独立的线程和执行器本身。
- ScheduledExecutorService，是 ExecutorService 接口的子接口，支持将来和/或周期性执行的任务。

通常，指向执行器对象的变量被声明为这三种接口类型中的一种，而不声明为执行器类类型。

在 Executor 接口中提供了一个单一的方法 execute()，被设计用来便捷地替代一个通用的线程创建用法。例如，如果 r 是一个 Runnable 对象，e 是一个 Executor 对象，那么可以将如下的代码：

```
(new Thread(r)).start();
```

替换为：

```
e.execute(r);
```

然而，execute()的定义是不太明确的。在低级的用法中，创建一个新的线程并立即启动它。而在 Executor 的实现中，execute()可能会这样做，但是也很有可能会使用一个已经存在的工作线程来运行 r，或者将 r 放在一个队列中等待有可用的工作线程。

在 java.util.concurrent 中实现的执行器被设计为充分利用 ExecutorService 和 ScheduledExecutorService 接口的最大好处，虽然这两个接口也使用基本的 Executor 接口工作。

ExecutorService 接口补充了一个与 execute()方法相似的但是更通用的 submit()方法。就像 execute()一样，submit()接受 Runnable 对象，但是也接受 Callable 对象，允许任务返回一个值。submit()方法返回一个 Future 对象，Future 对象用于获得 Callable 返回的值，并管理 Callable 和 Runnable 任务的状态。

ExecutorService 还提供用于提交大批量 Callable 对象的方法。最后，ExecutorService

提供许多用来管理执行器关闭的方法。要支持立即关闭，任务应该正确地处理 interrupts。

ScheduledExecutorService 接口为其父类补充了 schedule()方法，此方法在一个指定的延迟以后执行一个 Runnable 或 Callable 任务。另外，ScheduledExecutorService 接口还定义 scheduleAtFixedRate()和 scheduleWithFixedDelay()方法，它们按定义的时间间隔重复执行指定的任务。

2．线程池

大多数在 java.util.concurrent 包中实现的执行器使用线程池。线程池由工作线程组成。这种线程分别存在于它所执行的 Runnable 和 Callable 任务，并且经常用于执行复合任务。

使用工作线程可以将线程创建的负荷减小到最少。线程对象使用很大的内存资源，在大型的应用程序中，对大量线程对象的分配和撤销会生成很大的内存管理负荷。

线程池的一种最普通的类型是“固定线程池（fixed thread pool）”。这种类型的线程池总是维持一定数量的线程在运行；如果一个线程在使用期间因为某种原因而终止，它会自动地被一个新的线程所替代。任务通过一个内部队列被提交给线程池，只要有活动的任务超过线程数，这个内部队列就持有这些额外的任务。

使用线程池的一个很重要的好处是，应用程序使用它来实现“平稳下降”。要理解这个含义，考虑一个 Web 服务器应用程序。

在 Web 服务器应用程序中，每一个 HTTP 请求都由一个单独的线程来处理。如果应用程序只是简单地为每一个新的 HTTP 请求创建一个新的线程，那么当系统接收的请求超过它能够立即处理的数量时，如果这时这些线程超过了系统所能处理的能力，应用程序会突然停止对所有请求的响应。对所创建的线程加上数量上的限制，应用程序对 HTTP 请求的处理将不会像它们到来时那么快，但是它将会以系统能够维持的尽可能快的速度服务。

使用一个固定线程池创建一个执行器的简单方式是调用 java.util.concurrent.Executors 类的 newFixedThreadPool()工厂方法。java.util.concurrent.Executors 类还提供有以下的工厂方法：

- newCachedThreadPool()方法，使用一个可扩展的线程池创建执行器。这种执行器适合于启动许多短期任务的应用程序。
- newSingleThreadExecutor()方法，创建一个在同一时刻只执行一个单一任务的执行器。
- 上述执行器的 ScheduledExecutorService 版本的工厂方法。

如果上述工厂方法所提供的执行器都不符合需要，也可以通过构造 java.util.concurrent.ThreadPoolExecutor 的实例或 java.util.concurrent.ScheduledThreadPoolExecutor 的实例来创建满足特殊要求的执行器。

21.7.3 并发集合

在 java.util.concurrent 包中包括有许多额外的 Java 集合框架，这些集合可以根据其提供的集合接口加以分类：

- BlockingQueue，定义了一个先进先出的数据结构，当试图向一个满的队列中添加元素，或从一个空的队列中获取元素时，该队列会阻塞或超时。
- ConcurrentMap，是 java.util.Map 的子接口，定义了一些很有用的原子。这些操作只有在键存在的情况下才移除或替换存在的键值对，或者只有在健不存在的情况



下才添加键值对。原子化这些操作可以避免使用同步。对 ConcurrentMap 的标准通用的实现是 ConcurrentHashMap，这是一个 HashMap 的并发版。

- ConcurrentNavigableMap，是 ConcurrentMap 的子接口，支持模糊匹配。对 ConcurrentNavigableMap 的标准通用实现是 ConcurrentSkipListMap，这是 TreeMap 的一个并发版本。

所有这些集合帮助避免内存一致性错误，这是通过在一个向集合中添加一个对象的操作和后续的访问或移除对象的操作之间定义一个 happens-before 关系实现的。

【例 21.10】使用并发集合改写生产者-消费者程序。

（1）改写生产者类 Producer2，使用 BlockingQueue 对象作为共享队列。

```
package chapter21;
import java.util.Random;
import java.util.concurrent.BlockingQueue;
public class Producer2 implements Runnable {
    private BlockingQueue<String> drop;           //声明 BlockingQueue 对象
    public Producer2(BlockingQueue<String> drop) { //在构造器中初始化 drop 对象
        this.drop = drop;
    }
    public void run() {                           //线程 run 方法
        String importantInfo[] = {“消息 1”,”消息 2”,”消息 3”,”消息 4”};
        Random random = new Random();             //创建随机数对象
        try {
            for (int i = 0; i < importantInfo.length; i++) {
                                                  //循环将消息放入 drop 队列
                drop.put(importantInfo[i]);
                Thread.sleep(random.nextInt(5000));//线程休眠一个随机时间
            }
            drop.put("over");                     //"生产"结束，放入结束的信号
        } catch (InterruptedException e) {}
    }
}
```

（2）改写消费者类 Consumer，同样使用 BlockingQueue 对象作为共享队列。

```
package chapter21;
import java.util.Random;
import java.util.concurrent.BlockingQueue;
public class Consumer2 implements Runnable {
    private BlockingQueue<String> drop;
    public Consumer2(BlockingQueue<String> drop) {
        this.drop = drop;
    }
    public void run() {
        Random random = new Random();
        try {                    //从中间消息队列中获取生产者放入的消息并输出
            for (String message = drop.take(); ! message.equals("over");
message = drop.take()) {
                System.out.format("收到的消息: %s%n", message);
                Thread.sleep(random.nextInt(5000));       //休眠一个随机时间
            }
        } catch (InterruptedException e) {}
    }
}
```

（3）在 ProducerConsumerExample 的 main 方法中，将共享变量的类型声明为 BlockingQueue。

```
package chapter21;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.SynchronousQueue;
public class ProducerConsumerExample {
    public static void main(String[] args) {
        BlockingQueue<String> drop = new SynchronousQueue<String> ();
                                                              //创建同步队列
        (new Thread(new Producer2(drop))).start();            //启动生产者线程
        (new Thread(new Consumer2(drop))).start();            //启动消费者线程
    }
}
```

（4）使用如下命令编译 ProducerConsumerExample.java 程序：

```
javac -d . Producer2.java
javac -d . Consumer2.java
javac -d . ProducerConsumerExample.java
```

（5）最后，使用如下命令运行 ProducerConsumerExample 程序：

```
java chapter21.ProducerConsumerExample
```

输出结果如下所示。

```
收到的消息：消息 1
收到的消息：消息 2
收到的消息：消息 3
收到的消息：消息 4
```

代码说明：在 java.util.concurrent.BlockingQueue 接口中，定义了一个 get()方法，如果队列为空，该方法阻塞；还定义了一个 put()方法，如果队列全满，该方法为空。这与例 21.2 中的 Drop 类的操作相同。只不过 Drop 类不是一个队列。

不过也可以将 Drop 类视为容量为 0 的队列。因为在队列中没有任何元素的空间，每一个 get()方法会阻塞，直到相应的 take()发生，每个 take()方法也会阻塞，直到相应的 get()方法发生。在 java.util.concurrent 包中，有实现 BlockingQueue 接口的类 java.util.concurrent.SynchronousQueue，其行为与 Drop 类非常相似。

在本例中，使用 BlockingQueue 接口和 SynchronousQueue 类，因为其 put()和 get()方法会抛出 InterruptedException 异常，所以要将其放在 try-catch 语句块中。

BlockingQueue 不接受 null 元素。试图 add()、put()或 offer()一个 null 元素时，某些实现会抛出 NullPointerException。所以 null 也经常用做指示操作失败的警戒值。

21.7.4 原子变量

在 java.util.concurrent.atomic 包中定义的类支持在单一变量上的原子操作。所有的类都有 get()和 set()方法，它们就像在 volatile 变量上进行读写操作一样。这是因为，一个 set()方法与任何在同一变量上后续的 get()方法间都会建立一个 happens-before 关系。原子的 compareAndSet()方法还具有内存一致性特性，就像应用到整型原子变量上的简单原子算术方法。下面使用 java.util.concurrent.atomic 包改写 Counter 类。



【例 21.11】使用原子变量避免线程冲突。

```
package chapter21;
import java.util.concurrent.atomic.AtomicInteger;
class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
                                                    //创建具有初始值为 0 的原子变量
    public void increment() {
        c.incrementAndGet();                        //以原子方式将当前值加 1
    }
    public void decrement() {
        c.decrementAndGet();                        //以原子方式将当前值减 1
    }
    public int value() {
        return c.get();                             //获取原子变量当前值的方法
    }
}
```

在上面的程序中，AtomicInteger 类用在应用程序中以原子方式增加的计数器中。



21.8　实例：模拟银行系统进行存取款处理

并发程序设计在实际开发过程中，用到的也非常多，特别是并发业务比较多的情况下。在本节中，通过模拟银行系统存取款处理的实例，深入理解并发程序设计在实际开发中的应用。

【例 21.12】模拟银行系统进行存取款处理，要求存款和取款必须同步。实现步骤如下。

① 创建要进行存取款的账户类 Account，账户初始余额为 1000 元。在账户类 Account 中，定义一个同步的取款方法 withdraw()和一个同步的存款方法 deposit()。Account 类的定义如下所示。

```
class Account{                                      //创建 Account 类，代表银行的账户
    protected int balance=1000;                     //定义账户余额
    public synchronized void withdraw(int value){       //取钱同步方法
        balance=balance-value;                      //从账户余额中减去要取的款项
    }
    public synchronized void deposit(int value){        //存钱同步方法
        balance=balance+value;                      //向账户余额中增加存款
    }
    public int balance(){                           //返回账户当前余额
        return balance;
    }
}
```

② 创建一个单独的存款线程类 PersonA。该类的构造器带有一个 Account 类型的参数，代表要对其进行存款操作的账户。在该线程类中实现对传入的账户存款的功能。PersonA 类的定义如下。

```
class PersonA extends Thread{   //创建 PersonA 类代表存款人 A
     private Account acc;       //存款人 A 具有账户 Account，通过对象 acc 调用
     public PersonA(Account a){ //将存款人的账户作为参数传递进来
         acc=a;
     }
     public void run(){         //在此线程中，多次存款并输出账户余额信息
```

```
            for(int i=1;i<=2;i++){
                System.out.println();
                System.out.println("此时账户还有"+acc.balance()+"元");
                acc.deposit(200*i);                    //每次存入 200*i 元
                System.out.println("存款人 A 存入"+(200*i)+"元");
                System.out.println("此时账户还有"+acc.balance()+"元");
                try{
                   sleep(100);                         //此线程休眠 0.1 秒
                }catch(InterruptedException e) {}
            }
        }
    }
```

③ 再创建另一个单独的取款线程类 PersonB。该类的构造器带有一个 Account 类型的参数，代表要对其进行取款操作的账户。在该线程类中实现对传入的账户取款的功能。PersonB 类的定义如下。

```
class PersonB extends Thread{  //创建 PersonB 类代表存款人 B
     private Account acc;      //取款人 B 具有账户 Account，通过对象 acc 调用
     public PersonB(Account a){//将取款人的账户作为参数传递进来
         acc=a;
     }
     public void run(){        //在此线程中，多次取款并输出账户余额信息
         for(int i=1;i<=2;i++){
             System.out.println();
             System.out.println("此时账户还有"+acc.balance()+"元");
             acc.withdraw(100/i);              //每次取出 100/i 的金额
             System.out.println("存款人 B 取出"+(100/i)+"元");
             System.out.println("此时账户还有"+acc.balance()+"元");
             try{
                sleep(100);                    //此线程休眠 0.1 秒
             }catch(InterruptedException e) {}
         }
     }
}
```

④ 编写 BankDemo 类，在该类的 main()方法中，创建一个账户 a 以及对此账户进行存款操作的线程 tA 和对此账户进行取款操作的线程 tB。BankDemo 类的定义如下。

```
public class BankDemo {
   public static void main(String args[]){
      System.out.println("银行账户原有 1000 元");
      Account a = new Account();               //创建一个账户
      PersonA tA = new PersonA(a);             //创建存款人 tA
      PersonB tB = new PersonB(a);             //创建存款人 tB
      tA.start();                              //启动 tA 的线程
      tB.start();                              //启动 tB 的线程
   }
}
```

⑤ 在上面的程序中，创建了一个存款的线程和一个取款的线程。这两个线程分别同步地对同一个账户 Acount 对象 acc 进行存取款操作。使用如下的命令编译并运行此程序：

```
javac BankDemo.java
```



```
java BankDemo
```

程序运行以后，输出结果如下所示。

```
银行账户原有 1000 元

此时账户还有 1000 元
存款人 A 存入 200 元
此时账户还有 1200 元

此时账户还有 1200 元
存款人 B 取出 100 元
此时账户还有 1100 元

此时账户还有 1100 元
存款人 A 存入 400 元
此时账户还有 1500 元

此时账户还有 1500 元
存款人 B 取出 50 元
此时账户还有 1450 元
```

在上面的程序示例中，模拟了两个人（假设存款人 A 和取款人 B）在同一时间对同一个账号进行操作。账号本身有 1000 元，A 进行了两次存款操作，第一次存入 200 元，第二次存入 400 元；与此同时，B 进行了两次取款操作，第一次取 100 元，第二次取 50 元。如果 A 和 B 的操作是相对独立的话，那么 A 最终看到账号里有 1000+200+400=1600 元，而 B 最终看到账号里有 1000-100-50=850 元，很显然同一个账号不可能同时具有不同的金额，银行系统必须保证在同一时间内只有一个人可以对账号进行操作，这就需要使用线程同步实现。



21.9　小结

本章介绍了 Java 线程的一些基本知识和简单应用，并进一步讲解了并发程序中几个重要的概念：同步、死锁及高级并发对象。本章的内容对于初学者来说有一定的难度，特别是同步、死锁、不可改变对象及高级并发对象等。建议读者把学习的重点放在线程的理解及两种线程的实现方法上。其他并发内容，可以在将来从事相关开发工作时再做细致的研究。下一章将介绍如何使用 Socket 进行网络编程。

第22章 网络编程

Java 语言的发展和壮大得益于网络的发展，它最成熟的应用领域也是基于网络的企业级应用。可以说，Java 语言本身具有天然的网络特性。Java 提供了许多服务于网络的类库，使用户可以用流畅和完善的方式实现网络编程，完成各种复杂的网络应用开发。



22.1 网络编程基础

虽然 Java 中提供了丰富的网络相关类支持编写网络程序，而不直接使用网络底层协议，但是要开发网络应用程序，必须对网络的基础知识有一定的了解。Java 的网络通信可以使用 TCP、IP 和 UDP 等协议。在真正进行 Java 网络编程之前，对这些协议进行简单的介绍。

22.1.1 TCP 协议

TCP（Transmission Control Protocol），指的是传输控制协议，它是网络传输层的协议，主要负责数据的分组和重组。TCP 协议提供了一种可靠的数据传输服务，它是面向连接的，大多数的网络应用程序都使用 TCP 协议来实现传输层。使用 TCP 协议创建一个网络应用程序非常容易，它可以保证数据传送的时间、顺序和内容的正确无误。但是使用 TCP 需要大量的网络开销，所以如果希望实现更高效的传输，使用 TCP 就不适合了。

22.1.2 IP 协议

IP（Internet Protocol），又称为网际协议，它是网络层的协议，是比 TCP 低级的协议。它代表每个计算机在网络中的唯一标识。IP 地址是一个 32 位（IPv4）或 128 位（IPv6）的无符号数字，使用 4 组数字表示一个固定的编号，数字之间用一个点号隔开，例如“192.168.1.52”就代表网络中一个计算机唯一的地址编号。

IP 地址只是 IP 协议的一部分，IP 协议是一个无连接、不可靠的协议。Internet 上的所有数据都是通过 IP 包流动的，IP 包是 IP 传输的基本单位。

22.1.3 UDP 协议

UDP（User Datagram Protocol），指的是用户数据报协议。它和 TCP 协议一样，都是网



络传输层上的协议，但是它与 TCP 有着本质的区别。UDP 是一个不可靠的、无连接的和面向消息的协议，它使用数据报进行传输。使用 UDP 协议传输时，不保证数据一定能到达目的地，也不保证到达的顺序性。但是 UDP 协议占用资源比较少，所以一般用在一些可靠性要求比较低的网络应用上，如网络视频会议、在线影视和聊天室等音频、视频数据传送。

22.1.4　端口

端口（Port）可以被理解成计算机与外界通信交流的窗户。网络上的一台计算机可以提供多个服务，例如 Web 服务、FTP 服务和 Telnet 服务。那么，如何区分这些服务呢？单纯依靠 IP 地址是不行的，因为同一台计算机的 IP 地址是同一个。实际上，可以通过“IP 地址+端口号”的形式来区分不同的服务。当一个信息到达时，根据其请求的端口号不同，就可以知道应该提供哪个服务了。表 22.1 列举了一些常见服务的端口。

表 22.1　常见服务的端口

服　　务	端　　口
HTTP	80
FTP	21
Telnet	23
SMTP	25

端口号是一个整数，通常范围在 0～65535（即 16 位无符号整数）。小于 1024 的端口号一般是分配给特定的服务协议的，例如：TELNET（远程登录）、SMTP（简单邮件传输协议）、HTTP 或 FTP 等。所以如果是自定义的端口号，应尽量选择大于 1024 的整数。

22.1.5　套接字

套接字（Socket）可以理解为两个程序之间进行通信连接的两个终点接口。某个程序将一段信息写入套接字中，该套接字就会将这段信息发送给另外一个套接字，就像电话线的两端一样，这样另一端的程序就通过另一端的套接字收到了这段信息。所以使用套接字编程有时也称为 Socket 编程。

22.1.6　java.net 包

在 Java 的 API 中，java.net 包是被用来提供网络服务的。java.net 包中含有各种专门用于开发网络应用程序的类，程序开发人员可以使用该包中的类很容易地建立基于 TCP 可靠连接的网络程序，以及基于 UDP 不可靠连接的网络程序。java.net 包可以大致分为两个部分：

- 低级 API，用于处理网络地址（也就是网络标识符，如 IP 地址）、套接字（也就是基本双向数据通信机制）和接口（用于描述网络接口）。
- 高级 API，用于处理 URI（表示统一资源标识符）、URL（表示统一资源定位符）、URLConnection 连接（表示到 URL 所指向资源的连接）等。

22.2　使用 InetAddress 类封装 IP 地址

任何一台运行在 Internet 上的主机都有 IP 地址和当地 DNS 能够解析的域名。而在

java.net 包中相应提供了 IP 地址的封装类 InetAddress。

22.2.1 InetAddress 类

InetAddress 类用于描述和包装一个 Internet IP 地址，并提供了相关的常用方法，例如解析 IP 地址的主机名称、获取本机 IP 地址的封闭、测试指定 IP 地址是否可达等。InetAddress 实现了 java.io.Serializable 接口，不允许继承。它通过三个方法返回 InetAddress 实例。

- getLocalhost()：返回封装本地地址的实例。
- getAllByName(String host)：返回封装 Host 地址的 InetAddress 实例数组。
- getByName(String host)：返回一个封装 Host 地址的实例。其中，Host 可以是域名或者是一个合法的 IP 地址。

表 22.2 列出了 InetAddress 类的常用方法。

表 22.2 InetAddress 类的常用方法

方法名称	方法说明	返回类型
getLocalHost()	返回本地主机的 InetAddress 对象	InetAddress
getByName(String host)	返回指定主机名称的 IP 地址	InetAddress
getAllByName(String host)	返回指定主机名称数组	InetAddress 数组
getHostName()	获取本地主机名称	String
getHostAddress()	获取本地主机 IP 地址	String
isReachable(int timeout)	在指定的时间（毫秒）内，测试 IP 地址是否可到达	boolean

22.2.2 实例：获取本地主机和 Internet 公共域名的名称和 IP 地址

【例 22.1】编写一个应用程序的示例。程序的功能包括获取本地主机域名和 IP 地址；获取 Sun 公司网站的主机域名和 IP 地址。实现此程序的步骤如下。

① 在 D 盘根目录下，创建文件夹 chapter22，并在 chapter22 目录下，创建 java 源文件 InetAddressDemo.java；

② 使用记事本，打开 InetAddressDemo.java 源文件进行编辑，输入以下代码：

```
package chapter22;      //创建 chapter18 包
import java.net.*;
```

因为程序中使用了 net 类，所以要引入 java.net 包。

③ 接着编写 InetAddressDemo 类，并在该类中定义两个方法，分别用来获取本地主机域名和 IP 地址及 Sun 公司网站域名和 IP 地址。代码如下。

```
public class InetAddressDemo{
    //获取本地主机名和 IP 地址
    static void outputLocalHostAddress(){
        ……
    }
    //获得给定域名和 IP 地址
    static void outputIpAddress(){
        ……
    }
}
```



④ 实现方法 outputLocalHostAddress()，获得本机主机名和 IP 地址。具体代码如下。

```
//获取本地主机域名和 IP 地址
static void outputLocalHostAddress(){
    try{
        //返回运行程序主机的域名和 IP 地址
        InetAddress address = InetAddress.getLocalHost();
        System.out.println("本机的主机名为：" + address.getHostName());
        System.out.println("本机的 IP 为：" + address.getHostAddress());
    }catch(UnknownHostException e){
        //不能发现主机时抛出异常
        System.out.println("找不到主机地址。");
    }
}
```

⑤ 实现方法 outputIpAddress()，获得 Sun 公司网站域名和 IP 地址。具体代码如下。

```
//获得给定主机名和 IP 地址
static void outputIpAddress(){
    try{
        InetAddress address = InetAddress.getByName("www.sun.com");
        //输出主机名
        System.out.println("Sun 公司网站的域名为：" + address.getHostName());
        System.out.println("Sun 公司网站的 IP 为：" +
address.getHostAddress());
    }catch(UnknownHostException e){
        //不能发现主机时抛出异常
        System.out.println("找不到主机地址。");
    }
}
```

⑥ 编写 main()方法，调用上面定义的两个方法。代码如下。

```
public static void main(String[] args){
    outputLocalHostAddress();   //输出本机主机名和 IP 地址
    System.out.println();           //空行
    outputIpAddress();          //输出 Sun 公司网站域名的 IP 地址
}
```

⑦ 使用下面的命令编译并运行主程序 InetAddressDemo.java。

```
javac -d . InetAddressDemo.java
java chapter22. InetAddressDemo
```

⑧ 编译并运行以后，会看到如下的输出结果：

```
本机的主机名为：xlw
本机的 IP 为：192.168.1.52

Sun 公司网站的域名为：www.sun.com
Sun 公司网站的 IP 为：72.5.124.61
```



22.3　基于 URL 的网络编程

在 Java 类库中，提供了许多高级别的网络类。在使用这些类进行网络编程时，不需要

考虑底层实现的细节，这为程序开发提供了很大的便利。其中 URL（统一资源定位符）类就是这样的高级网络类，使用它可以方便地进行 URL 网络编程，访问 Web 资源。

22.3.1 URL 类

URL 表示 Internet 上某一资源的地址。通过 URL，开发人员可以访问 Internet 上的各种资源，例如最常见的 WWW 服务或者 FTP 服务。浏览器通过解析 URL，就可以找到相对应的资源。例如，下面是 Sun 公司的网站 URL：

```
http:// www.sun.com
```

这个 URL 由两部分组成：协议标识和资源名称。其中“http”为使用的协议，它指的是超文本传输协议（HTTP）。其他常用的协议还包括有文件传输协议（FTP）、Gopher、File 和 News。该 URL 中的“www.sun.com”为资源名称。

在 java.net 包中，提供了 URL 类对应 URL 地址。URL 类有多个构造器，可以用来创建一个 URL 实例。例如，下面的代码创建一个绝对 URL：

```
URL sinaURL = new URL("http://www.sina.com.cn/");
```

绝对 URL 包含所有目标 URL 的信息。包括主机名、文件名、端口号和资源引用。也可以创建一个相对的 URL。相对 URL 指的是只包含有相对另一个 URL 的资源的 URL。例如，有 http://www.sina.com.cn/news/today.html，那么可以用下面的代码创建该地址的相对 URL：

```
URL newsURL = new URL("http://www.sina.com.cn/news/");
URL todayNewsURL = new URL(newsURL,"today.html");
```

还可以创建一个 URL，指向网上某个网页内的锚点。例如，在 http://www.sina.com.cn/news/todays.htm 中有一个名为“BOTTOM”的锚点，位于网页底部。可以使用下面的代码创建一个指向 BOTTOM 锚点的相对 URL：

```
URL todayNewsBottomURL = new URL(todayNewsURL,"#BOTTOM");
```

此外，还有另外两种创建 URL 对象的方式，在某些时候比较有用。如下代码所示。

```
new URL("http",www.sina.com.cn", "/news/today.html");
new URL("http",www.sina.com.cn", 80,"news/today.html");
```

第一种构造器中各参数分别代表 URL 协议、主机名、资源文件名；第二种构造器多了一个端口号。这两种构造器与以下的构造方式等价：

```
new URL("http://www.sina.com.cn/news/today.html");
new URL("http://www.sina.com.cn:80/news/today.html");
```

使用这些构造器创建 URL 对象以后，可以使用 URL 对象的 toString()方法或等价的 toExternalForm()方法，获得包含全部 URL 地址的 String 字符串。

如果 URL 地址中有一个或多个空格，例如 http://myweb/hello world/，要使其合法，需要对空格进行编码，使用如下的代码：

```
URI uri = new URI("http","myweb","/hello world/","");
URL url = uri.toURL();                          //将 URI 转换为 URL
```

在上面的代码中，首先创建统一资源标识符，它会对资源地址中的空格自动编码；然后使用 URI 的 toURL()方法，将统一资源标识符转换为统一资源定位符。



在所有的 URL 构造器中，当参数为 null 或未知的协议时，都会抛出 MalformedURLException 异常，这通常需要使用 try/catch 语句来进行异常处理，如下列代码所示。

```
try {
    URL myURL = new URL(. . .)
} catch (MalformedURLException e) {
    . . .
    //这里为异常处理代码
    . . .
}
```



一旦创建了 URL 对象以后，就不能改变其属性（如协议、主机名、文件名或端口号）。

URL 类提供有多个方法，可以获取 URL 对象的协议、主机名、端口号、路径、查询字符串、文件名及资源引用。URL 的常见方法如表 22.3 所示。

表 22.3　InetAddress 类的常用方法

方法名称	方法说明
getProtocol()	获得该 URL 的协议名
getAuthority()	获得该 URL 的主机名和端口号
getFile()	获得该 URL 的文件名
getHost()	获得该 URL 的主机名
getPath()	获得该 URL 的路径
getPort()	获得该 URL 的端口号
getQuery()	获得该 URL 的查询字符串
getRef()	获得该 URL 的引用锚记名

例 22.2 是使用 URL 对象及其各种方法的例子。

【例 22.2】编写一个应用程序的示例。程序的功能为使用 URL 的各种方法。

（1）编写 ParseURLDemo 类及其 main()方法。代码如下。

```
package chapter22;
import java.net.*;
import java.io.*;
public class ParseURLDemo {
    public static void main(String[] args) throws Exception {
        URL aURL = new URL("http://java.sun.com:80/docs/books/tutorial"
                           + "/index.html?name=networking#DOWNLOADING");
        System.out.println("protocol = " + aURL.getProtocol());
                                              //输出 URL 对象的协议
        System.out.println("authority = " + aURL.getAuthority());
                                              //输出 URL 对象的主机各和端口号
        System.out.println("host = " + aURL.getHost()); //输出 URL 的主机名
        System.out.println("port = " + aURL.getPort()); //输出 URL 的端口号
        System.out.println("path = " + aURL.getPath()); //输出 URL 的路径
        System.out.println("query = " + aURL.getQuery());  //输出URL的查询字符串
        System.out.println("filename = " + aURL.getFile()); //输出 URL 的文件名
        System.out.println("ref = " + aURL.getRef()); //输出 URL 的引用的锚记名
    }
```

```
}
```

（2）编译并运行此程序，代码如下。

```
javac -d . ParseURLDemo.java
java chapter22. ParseURLDemo
```

（3）程序的输出结果如下。

```
protocol = http
authority = java.sun.com:80
host = java.sun.com
port = 80
path = /docs/books/tutorial/index.html
query = name=networking
filename = /docs/books/tutorial/index.html?name=networking
ref = DOWNLOADING
```

对比输出结果和程序代码，体会各个方法的作用。需要注意的是，在 ParseURLDemo 类的 main()方法声明中，抛出异常，由系统处理，所以在代码中没有使用 try/catch 语句处理异常。

22.3.2 直接读取 URL 内容

在创建了 URL 对象以后，就可以使用 URL 的 openStream()方法来获得一个输入流，使用这个输出流可以直接读取 URL 中的内容。URL 的方法 openStream()返回一个 java.io.InputStream 对象，所以从 URL 中读取内容与从一个输入流中读取内容是一样的。

下面的例 22.3 中，程序使用 openStream()方法打开一个 http://www.163.com 的 URL 的输入流。然后再使用 BufferedReader 流过滤，最后将从流中读取的数据输出到标准输出流中。

【例 22.3】编写一个应用程序的示例。程序的功能为读取 URL 中的内容。

（1）编写 URLReaderDemo 类及其 main()方法。代码如下。

```
package chapter22;
import java.net.*;
import java.io.*;
public class URLReaderDemo{
    public static void main(String[] args) throws Exception {
        URL wy = new URL("http://www.163.com/");   //创建 URL 对象
        BufferedReader in = new BufferedReader(
                    new InputStreamReader(
                    wy.openStream()));              //打开输入流
        String inputLine;
        while ((inputLine = in.readLine()) != null)    //循环读取输入流 in
中的每一行内容
            System.out.println(inputLine);      //将读取的内容输出到控制台
        in.close();                             //关闭流
    }
}
```

（2）编译并运行此程序，代码如下。

```
javac -d . URLReaderDemo.java
java chapter22. URLReaderDemo
```



（3）程序的输出结果为网易首页的 HTML 代码。

需要注意的是，在 ParseURLDemo 类的 main()方法声明中，抛出异常，由系统处理，所以在代码中没有使用 try/catch 语句处理异常。

22.3.3　URLConnection 类

通过 URL 类的 OpenStream()方法，只能从网络上读取数据，如果想同时输出数据，则必须先与 URL 建立连接，然后才能对其进行读写，这时就要用到 URLConnection 类了。

URLConnection 类也位于 java.net 包中，调用 URL 类的 openConnection()方法可以获得 URLConnection 对象。获得对象后，开发人员可以调用该对象的 connect()方法来连接远程资源，如下代码所示。

```
try {
    URL wy = new URL("http://www.163.com/");
    URLConnection wyConnection = wy.openConnection(); //获得URLConnection对象
    wyConnection.connect();                           //连接远程资源
} catch (MalformedURLException e) {              //创建URL失败时,会产生此异常
    . . .
} catch (IOException e) {          //openConnection() 方法失败时,产生此异常
    . . .
}
```

和远程资源建立连接以后，就可以执行相关操作，包括查询 HTTP 请求头信息、访问资源数据等，以及写入相关内容等。

并不总是需要显式地调用 connect()方法来与远程资源建立连接。当使用 URLConnection 类的 getInputStream()或 getOutputStream()方法时，会隐含地建立连接。

下面的例 22.4 与上一节的例 22.3 的作用相同，但是是使用 URLConnection 对象并调用其 getInputStream()方法从连接获得一个输入流，然后程序在此输入流上创建 BufferedReader 流并读取内容。

【例 22.4】编写一个应用程序的示例。程序的功能为读取 URL 中的内容。

（1）编写 URLConnectionReaderDemo 类及其 main()方法。代码如下。

```
package chapter22;
import java.net.*;
import java.io.*;
public class URLConnectionReaderDemo{
    public static void main(String[] args) throws Exception {
        URL wyURL = new URL("http://www.163.com/"); //创建URL对象
        URLConnection con = wyURL.openConnection();    //获得URLConnection对象
        BufferedReader in = new BufferedReader(
                            new InputStreamReader(
                            con.getInputStream())); //从连接对象获得输入流
        String inputLine;
        while ((inputLine = in.readLine()) != null) //从输入流一行行读取 URL
的内容
            System.out.println(inputLine);           //输出到控制台
        in.close();                                  //关闭输入流
    }
}
```

（2）编译并运行此程序，代码如下。

```
javac -d . URLConnectionReaderDemo.java
java chapter22. URLConnectionReaderDemo
```

（3）程序的输出结果为网易首页的 HTML 代码，与例 22.3 相同。

需要注意的是，在 main 方法声明中抛出异常，由系统处理，所以在代码中没有使用 try/catch 语句处理异常。

虽然读取内容时与 URL 的 openStream()方法相同，但是使用 URLConnection 类的 getOutputStream()方法获得输出流，还可以向输出流中写入数据，在 URL 另一端的服务器程序可以接收输入的数据。

22.4 基于 TCP 的网络编程

TCP（传输控制协议）是一种基于连接的协议，可以在计算机之间提供可靠的数据传输。使用 TCP 传输数据就好比打电话，需要先建立连接通道，再传送数据。连接通道的两端通常称为套接字（Socket）。套接字分为客户端套接字和服务器端套接字，就好像在打电话时所需要的通话两端的电话筒一样。在拨通电话之后，就可以通过电话筒与对方通话，也可以听到对方所说的话。基于 TCP 的网络通信也是如此，先建立起连接，再通过套接字（Socket）发送数据和接收数据。

22.4.1 Socket 通信模型

通过 TCP 进行通信的双方通常称为服务器端和客户端。服务器端和客户端可以是两台不同的计算机，也可以是同一台计算机。服务器端运行的是服务器端程序，和运行在客户端的客户端程序有所不同。通过 TCP 进行网络数据通信的模型如图 22.1 和图 22.2 所示。



图 22.1　连接请求



图 22.2　建立 Socket 连接

服务器和客户端进行通信的过程如下。

（1）服务器端程序启动，监听客户端连接请求。

（2）客户端程序向服务器端程序发出连接请求。

（3）服务器端接收到连接请求以后，创建一个带有指定端口的 Socket，与客户端建立一个 Socket 连接。

（4）服务器端和客户端通过建立的 Socket 连接通道通信（传输数据）。



在 java.net 包中，有一个 Socket 类，用来建立 Socket 套接字。java.net 包中还有一个 ServerSocket 类，用在服务器端，其有一个 accept 方法，用来监听和接收客户端的连接请求。

java.net.Socket 类是开发网络应用程序的服务器端和客户端程序都要用到的套接字类。它提供了许多方法用来获得相关的信息，如表 22.4 所示。

表 22.4　Socket 类的常用方法

方法名称	方法说明
getInputStream()	从套接字中获得输入流
getInetAddress()	从套接字中获得网络地址
getLocalAddress()	获得本机的网络地址
getPort()	获得套接字中的端口
getLocalPort()	获得本机端口
getOutputStream()	从套接字中获得输出流
close()	关闭套接字

22.4.2　服务器端程序设计

服务器端程序需要用到 java.net.Socket 类和 java.net.ServerSocket 类。服务器端程序的建立通常由如下 5 个步骤组成：

① 在服务器端程序中，首先创建类 java.net.ServerSocket 的实例对象，注册在服务器端进行连接的端口号以及允许连接的最大客户数目。

② 调用 ServerSocket 的成员方法 accept()，等待并监听来自客户端的连接。当有客户端与该服务器端建立起连接时，accept()方法将返回 Socket 连接通道在服务器这端的套接字（Socket）。通过该套接字可以与客户端进行数据通信。

③ 调用服务器端套接字 Socket 的方法 getInputStream()和 getOutputStream()，获得该套接字所对应的输入流(InputStream)和输出流(OutputStream)。

④ 通过获得的输入流和输出流与客户端进行数据通信，并处理从客户端获得的数据及需要向客户端发送的数据。

⑤ 在数据传输结束以后，关闭输入流、输出流和套接字。

在服务器端创建 ServerSocket 的实例对象，并调用其 accept()方法之后，服务器端开始一直等待客户端与其连接。ServerSocket 类的常用方法如表 22.5 所示。

表 22.5　ServerSocket 类的常用方法

方法名称	方法说明
accept()	阻塞程序等待客户端连接请求。一旦接收到连接请求，则返回一个表示连接已经建立的 Socket 对象
close()	关闭当前的 Socket
getInetAddress()	获得本机网络地址
getLocalPort()	获得监听端口

下面的例 22.5 是一个简单的网络程序的服务器端程序代码。

【例 22.5】编写一个简单网络应用程序的服务器端程序示例。程序的功能为读取监听客户端的连接请求，并向客户端发送问候信息。当客户发送过来“end”时，结束连接。

```
import java.io.*;
```

```
import java.net.*;
public class MyServerDemo {
    public static void main(String[] args) throws IOException{
        ServerSocket server =new ServerSocket(5678);    //创建ServerSocket
实例，参数为监听的端口号
        System.out.println(“服务器已启动，正在等待连接……”);
        Socket client=server.accept();                  //阻塞，等待客户端连接
        System.out.println(“客户端建立连接。”);
        //获得 Socket 连接的字节输入流并转换为缓冲字符流
        BufferedReader in = new BufferedReader(new InputStreamReader(client.
getInputStream()));
        //获得 Socket 连接的打印输出流
        PrintWriter out=new PrintWriter(client.getOutputStream());
        while(true){                                    //循环读取输入流中的内容
            String str=in.readLine();                   //读取输入流中的一行内容
            System.out.println(str);                    //将内容输出到控制台
            out.println("已接收....");                  //向输出流中输出信息
            out.flush();                                //刷新输出流
            if(str.equals("end"))       //如果客户端发过来“end”，说明结束连接
                break;
        }
        in.close();                                     //按顺序关闭输入输出流
        out.close();
        client.close();                                 //关闭 Socket 连接
        server.close();                                 //关闭 ServerSocket
    }
}
```

如上面的代码所示，首先要在服务器端使用 ServerSocket(int port)构造器构造一个 ServerSocket 的服务器端套接字实例。其中参数 port 为 ServerSocket 类要监听的端口。创建 ServerSocket 可以使用 4 种构造器，如下所示。

```
ServerSocket server1 = new ServerSocket();
ServerSocket server2 = new ServerSocket(5678);
ServerSocket server1 = new ServerSocket(5678,100);
ServerSocket server1 = new ServerSocket(int port,int backlog,InetAddress
bindAddr);
```

其中不带参数的构造器为默认构造方法，可以创建未绑定端口号的服务器套接字。带一个参数的构造器，将创建绑定到参数指定端口的服务器套接字对象，并且默认的最大连接队列长度为 50。如果连接数量超出 50 个，将不会再接收新的连接请求。带两个参数的构造器使用参数指定的端口号和最大连接队列长度创建服务器端套接字对象。而最后一个构造器，当服务器有多个 IP 地址时，使用 bindAddr 参数指定创建服务器套接字的 IP 地址。

如果服务器只有一个 IP 地址，那么没有必要使用最后一个构造器。所有的构造器都需要处理 IOException 异常。

在创建服务器端套接字 server 以后，使用该对象的 accept()方法接收客户端请求的套接字连接。例如：

```
Socket client = server.accept();
```

该方法被调用之后，将等待客户的连接请求，在接收到客户端的套接字连接请求以后，



该方法将返回 Socket 对象，这个 Socket 对象是已经和客户端建立好连接的套接字，可以通过这个 Socket 对象获取客户端的输入/输出流来实现数据发送与接收。accept()方法可能会产生 IOException 异常，因此在调用该方法时必须捕获并处理该异常。

accept()方法将阻塞当前线程，直到接收到客户端的连接请求为止。所以只有当客户的连接请求被接收以后，此语句后面的语句才会被执行。

在接收到客户端请求并与客户端建立 Socket 连接以后，就可以打开服务器端的 Socket 上的输入流和输出流，这用到了 Socket 类的两个方法：getInputStream() 和 getOutputStream()。通过这两个流，就可以像其他输入/输出流一样向其中写入或从其中读取数据了。

22.4.3　客户端程序设计

服务器端程序需要用到 java.net.Socket 类。在服务器/客户端的网络应用程序中，每次的 TCP 连接请求都会由客户端向服务器端发起。客户端程序的建立通常由如下 4 个步骤组成：

① 在客户端程序中，创建类 java.net.Socket 的实例对象，与服务器端建立起连接。在创建 Socket 的实例对象时需要指定服务器端的主机名以及进行连接的端口号（此端口号一定要与服务器端的 ServerSocket 构造实例对象时所监听的端口号一致）。

② 调用客户端套接字 Socket 的方法 getInputStream()和 getOutputStream()，获得该套接字所对应的输入流(InputStream)和输出流(OutputStream)。

③ 通过获得的输入流和输出流与服务器端进行数据通信，并处理从服务器端获得的数据及需要向服务器端发送的数据。

④ 在数据传输结束以后，关闭输入流、输出流和套接字。

例 22.6 实现了一个客户端程序连接到新浪服务器的 80 端口，并输出了服务器和客户机的信息。

【例 22.6】编写一个简单的 Socket 客户端程序示例。程序的功能为连接到新浪服务器的 80 端口，并输出了服务器和客户机的信息。

```
package chapter22;
import java.io.*;
import java.net.*;
public class WebClientDemo {
    public static void main(String[] args) throws IOException{
        Socket client = new Socket("www.sina.com.cn",80);
                                        //创建 Socket 实例，连接到新浪网的 80 端口
        //输出服务器端信息
        System.out.println("服务器 IP 是: " + client.getInetAddress());
        System.out.println("服务器端口号是: " + client.getPort());
        //输出客户机信息
        System.out.println("客户机 IP 是: " + client.getLocalAddress());
        System.out.println("客户机端口号是: " + client.getLocalPort());
        client.close();             //关闭 Socket 连接
    }
}
```

编译并运行此程序，输出结果如下。

```
服务器 IP 是：www.sina.com.cn/202.108.33.32
服务器端口号是：80
客户机 IP 是：/192.168.1.52
客户机端口号是：1682
```

例 22.7 实现了一个与例 22.5 服务器端程序相对应的客户端程序。

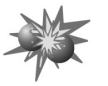
【例 22.7】编写一个简单的网络应用程序的服务器端程序示例。程序的功能为向服务器端发出连接请求，并向服务器端发送问候信息。当发送“end”字符串信息时，结束连接。

```
package chapter22;
import java.io.*;
import java.net.*;
public class MyClientDemo {
    public static void main(String[] args)throws Exception{
        //创建 Socket 实例，参数为要连接的服务器和端口号
        Socket server = new Socket(InetAddress.getLocalHost(),5678);
        //获得 Socket 连接的字节输入流并转换为缓冲字符流
        BufferedReader in=
                new     BufferedReader(new     InputStreamReader(server.
getInputStream()));
        //获得 Socket 连接的打印输出流
        PrintWriter out=new PrintWriter(server.getOutputStream());
        //控制台输入流转换为缓冲字符流
        BufferedReader   wt=new   BufferedReader(new   InputStreamReader
(System.in));
        while(true){
            String str=wt.readLine();           //从控制台接收输入
            out.println(str);                   //向服务器发送信息
            out.flush();                        //刷新输出流
            if(str.equals("end")){              //如果用户输入的是“end”
                break;                          //结束循环
            }
            System.out.println(in.readLine()); //接收服务器发送的信息并输出
        }
        in.close();                             //按顺序关闭输入输出流
        out.close();
        wt.close();
        server.close();                         //关闭套接字
    }
}
```

如上面的代码所示，首先使用 Socket 构造器构造套接字对象，其带有两个参数。第一个参数为 InetAddress 封装的 IP 地址（或要连接的服务器的 IP 地址），第二个参数为要连接的端口号。创建 Socket 对象可能会产生 UnknownHostException 和 IOException 异常，需要捕获和处理，或抛出。

建立 Socket 连接以后，就可以打开 Socket 连接上的输入流和输出流，这用到了 Socket 类的两个方法：getInputStream()和 getOutputStream()。通过这两个流，就可以像其他输入/输出流一样向其中写入或从其中读取数据了，从而实现与服务器端程序通信。

首先启动例 22.5 中的服务器应用程序（一定要先启动服务器端程序），使其监听客户端的连接。然后再打开一个控制台窗口，编译并运行例 22.7 中的客户端应用程序，并从控制台输



入要向服务器传送的信息。客户端和服务器端程序运行结果分别如图 22.3 和图 22.4 所示。

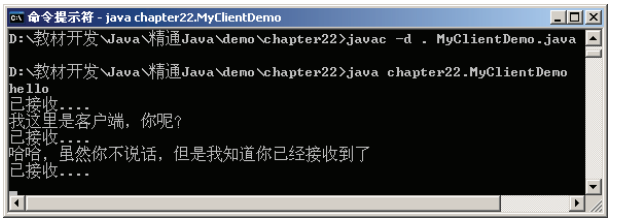


图 22.3　客户端程序



图 22.4　服务器端程序

22.4.4　多客户端同时访问服务器端程序

上面的服务器端程序和客户端程序实现的只是简单的两台计算机之间的通信。如果是多个客户同时访问一个服务器呢？试着再运行一个客户端，结果是会抛出异常的。那么多个客户端如何实现呢?

其实，简单地分析一下，就可以看出客户和服务通信的主要通道就是 Socket 本身。而服务器通过 accept()方法同意和客户建立通信。这样当客户建立 Socket 的同时，服务器也会使用这一根 Socket 连接来先后通信。那么既然如此，只要存在多条 Socket 连接就可以了。

多个客户端程序同时访问一个服务器，这很显然是一个并行执行的问题了，所以线程是最好的解决方案。

首先要做的事情是创建线程并使其可以和网络连接取得联系，然后由线程来执行刚才的操作。要创建线程，要么直接继承 Thread，要么实现 Runnable 接口，要建立和 Socket 的联系，只要传递引用就可以了。而要执行线程，就必须重写 run()方法。而 run()方法所做的事情，就是刚才单线程版本 main()所做的事情。因此将程序修改如下。

【例 22.8】编写一个简单的网络应用程序的服务器端程序示例。程序的功能为向服务器端发出连接请求，并向服务器端发送问候信息。当发送“end”字符串信息时，结束连接。服务器可以并发响应多个客户端程序的连接请求，并同时与多个客户端程序通信。

```
package chapter22;
import java.io.*;
import java.net.*;
public class ThreadServerDemo extends Thread{
    private Socket client;
    public ThreadServerDemo(Socket c){        //构造器，接收传入的 socket 连接引用
        this.client=c;
    }
    public void run(){
        try{
            //获得 Socket 连接的字节输入流并转换为缓冲字符流
```

```
            BufferedReader in = new BufferedReader(new InputStreamReader
(client.getInputStream()));
            //获得 Socket 连接的打印输出流
            PrintWriter out=new PrintWriter(client.getOutputStream());
            while(true){                          //循环读取输入流中的内容
                String str=in.readLine();         //读取输入流中的一行内容
                System.out.println(str);          //将内容输出到控制台
                out.println("已接收....");         //向输出流中输出信息
                out.flush();                      //刷新输出流
                if(str.equals("end"))   //如果客户端发过来“end”，说明结束连接
                    break;
            }
            client.close();              //关闭 Socket 连接
        }catch(IOException ex){
        }finally{}
    }
    public static void main(String[] args)throws IOException{
        ServerSocket server=new ServerSocket(5678);
        while(true){
            Socket socket = server.accept();//每当accept()返回一个Socket连接
            //启动一个新的线程，负责 socket 连接的通信
            ThreadServerDemo mu=new ThreadServerDemo(socket);
            mu.start();                       //启动线程的 run 方法
        }
    }
}
```

ThreadServerDemo 类直接从 Thread 类继承了下来，并且通过构造函数传递引用和客户 Socket 建立了联系。这样每个线程就有了一个通信管道。同样可以编写 run()方法，把之前的操作交给线程来完成，这样多客户并行的 Socket 就建立起来了。

22.5 基于 UDP 的网络编程

相对 TCP 而言，UDP 的应用并不那么广泛，几个主要的应用层协议如 HTTP、FTP 和 SMTP 等使用的都是 TCP 协议。但是随着计算机网络的发展，UDP 协议正逐渐显示它的优点。对于信息可以被分割成一些无关联的消息进行传输的情况，UDP 是一个非常好的选择，特别是在需要很强的实时交互性的场合，如网络游戏、视频会议、股票信息等。下面就介绍如何用 Java 来实现 UDP 网络编程。

22.5.1 UDP 通信模型

UDP 通信，又称为数据报通信。UDP 协议采用的是基于数据报的网络通信。数据报是一种分组交换的形式，就是把所有要传送的数据分段打成包，再传送出去。它属于无连接型，是把打成的每个包（分组）都作为一个独立的报文传送出去，所以叫做数据报。

UDP 通信与通过邮局寄信和取信的过程相似，在寄信之前，不需要和另一方建立专门的连接，只需要知道对方的地址和门牌号（相当于服务器地址和端口号），就可以把信寄出去。由于没有专门建立连接，所以不保证数据报包会顺利到达指定的主机，也不保证数据报包会按照发送的顺序到达指定的主机。



在 Java 的 UDP 网络程序中，主要用到两个类：java.net.DatagramSocket 和 java.net.DatagramPacket 类。其中 DatagramSocket 类用于发送和接收数据的数据报套接字，而 DatagramPacket 类是 UDP 所传递的数据报包，即打包后的数据。

DatagramPacket 类表示的数据报有两种，一种负责传送数据，一种负责接收数据。一般用来传送数据的数据报使用如下的构造器创建：

```
public DatagramPacket(byte[] buf,          //要打包的数据
                      int length,          //数据报包的长度
                      InetAddress address, //目的主机地址
                      int port);           //目的端口号
```

该构造器用于构造数据报包，将长度为 length 的包发送到指定主机上的指定端口号。其中 buf 参数表示的是需要传送的字节数据，lentgh 参数是所传送数据的长度，address 参数表示的是一个封装了目标地址的 InetAddress 对象，存储着接收者的 IP 信息。port 表示接收方的监听端口。length 参数必须小于等于 buf.length。当该数据报包被发送时，在数据报包中会自动添加上发送方的网络地址及其端口号。负责接收数据的数据报使用如下的构造器创建。例如：

```
public DatagramPacket(byte[] buf, int length );
```

其中 buf 参数表示用来接收传来数据的字节数组，length 参数是传来数据的长度。如果说 DatagramPacket 类的作用是表示一个数据报，那么，DatagramSocket 类的作用就是在程序之间建立传送数据报的通信连接。常用的 DatagramSocket 类的构造器有两种，一种是不带参数的构造器，一种是带有端口号参数的构造器。不带参数的构造器为默认的构造器，如下所示。

```
public DatagramSocket() throws SocketException;
```

该构造器使用本机任何可用的端口创建数据报套接字实例。在创建 DatagramSocket 类的实例时，有可能会产生 SocketException 异常。另一种常用的构造器为带有指定端口号的构造器，如下所示。

```
public DatagramSocket(int port) throws SocketException;
```

该构造器创建数据报套接字并将其绑定到 port 参数指定的本机端口，端口号取值必须在 0～65535 范围内（包括 0 和 65535）。

一般在接收数据的一方，使用带有端口号的 DatagramSocket 构造器，而在发送数据的一方，使用不带参数的 DatagramSocket 构造器。当接收方收到来自发送方的数据时，可以从该数据报包中得到发送方的网络地址（使用 DatagramPacket 类的 getAddress()方法）和端口号（使用 DatagramPacket 类的 getPort()方法）。因此，发送放程序可以通过不带参数的构造器创建数据报套接字对象。DatagramSocket 类常见的方法如表 22.6 所示。

表 22.6　DatagramSocket 类的常用方法

方法名称	方法说明
send(DatagramPacket p)	发送一个数据报
receive(DatagramPacket p)	接收一个数据报
disconnect()	断开这个 Socket
close()	关闭这个 Socket

22.5.2　接收数据报程序设计

要编写一个能接收数据报的程序，根据上一节的介绍可以知道，需要使用如下两个类。

一个是用来接收数据报的套接字对象（即 DatagramSocket），一个是用来接收数据报的数据报对象（即 DatagramPacket）。其中用来接收数据报的套接字对象使用如下带有指定端口参数的构造器创建，如下所示。

```
DatagramSocket  ds = new DatagramSocket(3256);
```

另外，需要创建一个数据报，来接收传送过来的数据报，如下所示。

```
byte[ ] datas = new byte[1024];     //定义一个字节数组来接收传送过来的数据
DatagramPacket data = new DatagramPacket(datas,datas.length); //定义一个用来接收传送过来数据的数据报
……
ds.receive(data);                   //将接收到的数据报放到 data 对象中
```

例 22.9 实现了一个接收数据报的简单程序。

【例 22.9】编写一个简单的 UDP 网络应用程序。程序的功能为：接收另一方传送过来的数据报信息，并输出。程序代码如下所示。

```
package chapter22;
import java.net.*;
public class PacketReceiverDemo{
    public static void main(String[] args) throws Exception{
        //定义一个字节数组来接收传送过来的数据
        byte[ ] datas = new byte[1024];
        //定义套接字对象
        DatagramSocket  ds = new DatagramSocket(3256);
        //定义一个用来接收传送过来数据的数据报
        DatagramPacket data = new DatagramPacket(datas,datas.length);
        //接收数据报包
        System.out.println("正在等待接收数据......");
        while(true){
            datas = new byte[1024];          //当循环接收数据时，要清空字节数组
            data.setData(datas);    //将新的空字节数组设为接收数据报的字节数组
            ds.receive(data);       //接收数据报包
            String message = new String(datas).trim();
                                    //将字节数组转换为字符串
            System.out.println(message);
            if(message.equals("end"))       //如果对方发过来“end”，结束循环
                break;
        }
        System.out.println("结束通话。");
        ds.close();                         //关闭套接字
    }
}
```

上面的程序使用一个端口号为 3256 的 DatagramSocket 接收数据报，而后通过一个 while 循环来接收各个客户端向这个程序发送的信息，并输出到控制台上。程序运行以后，这个程序会启动，循环等待接收数据。

22.5.3 发送数据报程序设计

要编写一个能发送数据报的程序，根据上一节的介绍可知，需要使用如下两个类。一个是用来发送数据报的套接字对象（即 DatagramSocket），一个是被用来发送的数据报对



象（即 DatagramPacket）。其中用来发送数据报的套接字对象使用不带参数的构造器创建，如下所示。

```
DatagramSocket  ds = new DatagramSocket();
```

这时，该构造器会自动为该数据报套接字查找并配置当前可用的端口号。在接收方，可以从接收到的数据报中获得发送方的网络地址和端口号。另外，需要创建一个数据报，来封装要发送的数据，如下所示。

```
InetAddress address = ……;                          //定义接收方主机地址
String message = “Hello,this is a UDP demo!”; //定义要发送的信息
byte[ ] datas = message.getBytes();                 //将信息字符串转换为字节数组
//定义一个用来发送的数据报
DatagramPacket data = new DatagramPacket(datas,datas.length,address,3256);
……
ds.send(data);                                      //发送数据报
```

例 22.10 实现了一个接收数据报的简单程序。

【例 22.10】编写一个简单的 UDP 网络应用程序。程序的功能为：向另一方程序发送数据报信息，并输出。程序的代码如下所示。

```
package chapter22;
import java.net.*;
import java.io.*;
public class PacketSendDemo{
    public static void main(String[] args) throws Exception{
        String local = "localhost";
        InetAddress address = InetAddress.getByName(local); //定义接收方主机
地址
        DatagramSocket  ds = new DatagramSocket();          //定义套接字
        //控制台输入流转换为缓冲字符流
       BufferedReader wt=new BufferedReader(new InputStreamReader(System.
in));
        System.out.println("请输入要发送的信息：");
        while(true){
            String message = wt.readLine();     //从控制台接收要发送的信息
            byte[ ] datas = message.trim().getBytes(); //将信息字符串转换为
字节数组
            //定义一个用来发送的数据报
            DatagramPacket data = new DatagramPacket(datas,datas.length,
address,3256);
            ds.send(data);                                  //发送数据报
            if(message.equals("bye"))
                break;
        }
        System.out.println("结束通话。");
        ds.close();                                         //关闭套接字
    }
}
```

上面的程序使用一个端口号为自动绑定的 DatagramSocket 来发送数据报，而后通过一个 while 循环来接收从控制台输入的要发送的信息。程序运行以后，这个程序会启动，循环等待接收数据并发送。

22.5.4 运行 UDP 网络程序

使用如下的步骤测试程序的运行：

① 打开控制台，使用如下的命令编译并运行 PacketReceiverDemo.java 程序：

```
javac -d . PacketReceiverDemo.java
java chapter22.PacketReceiverDemo
```

这时程序阻塞，等待接收数据。

② 另外再打开一个控制台窗口，使用如下的命令编译并运行 PacketSendDemo.java 程序：

```
javac -d . PacketSendDemo.java
java chapter22. PacketSendDemo
```

③ 在发送数据报的控制台中，输入要发送的信息，两个程序间的通信如图 22.5 所示。



图 22.5 使用 UDP 进行网络通信

22.6 小结

在本章中，学习了如何使用 Java 的 java.net 包所提供的类来进行 Socket 网络编程。重点讲了基于 TCP 协议的可靠的和面向连接的网络通信，以及基于 UDP 协议的不可靠的和无连接的网络通信。这是本章学习的重点。其中应用多线程处理多客户端的访问是本章学习的难点，读者应结合上一章线程部分的内容来理解这部分内容。下一章将介绍使用 Java 的 JDBC 技术进行 Java 数据库应用程序开发的基础知识。

第23章 Java数据库应用程序开发基础

软件的本质就是处理数据，而在商业应用中，数据往往需要专门的数据库来存放。所以在软件开发过程中，几乎所有的项目都要使用到数据库。那么在Java项目中是如何对数据库进行操作的呢？这就需要用到Java数据库应用程序。在本章中，将学习如下内容：

- JDBC技术和驱动程序；
- java.sql包下的常用类；
- JDBC操作数据库实例。

23.1 JDBC技术概述

JDBC（Java Database Connectivity）是一种用于执行SQL语句的Java API。它由一组用Java编程语言编写的类和接口组成，为Java应用程序与各种不同数据库之间进行对话提供了一种便捷的方法，使得开发人员能够用纯Java API来编写具有跨平台性的数据库应用程序。

23.1.1 JDBC技术介绍

应用程序可以通过数据库厂商提供的API及SQL语句对数据库进行操作，如图23.1所示。

在此模式下，看到不同的数据库有不同的API操作界面。这样的话，对于实现同样功能的应用程序，针对不同的数据库，开发人员需要编写不同的代码，这样对Java设计人员是很不方便的。于是JDBC应运而生。

JDBC的出现使Java程序对各种数据库的访问能力大大增强。通过使用JDBC，开发人员可以很方便地将SQL语句传送给几乎任何一种数据库。

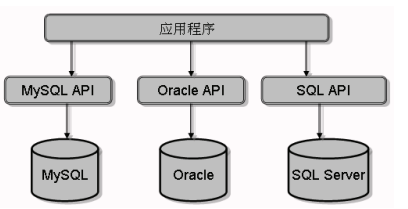


图23.1 应用程序访问数据库

JDBC 的工作模式如图 23.2 所示。从图中可以看到，应用程序通过调用 JDBC 来操作数据库的过程，其实是由数据库厂商提供的 JDBC 驱动程序来负责的。如果要更换数据库，只要更换驱动程序，并在 JDBC 中载入新的驱动程序来源，即可完成数据库系统的变更。换句话说，使用 JDBC 后就没有必要写一个程序访问 Oracle 数据库，而另一个程序访问 SQL Server 数据库，再写一个程序访问 DB2 了。

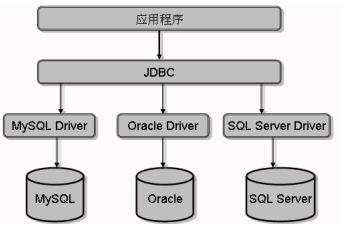


图 23.2　JDBC 工作模式

JDBC 的主要功能如下。

- 建立与数据库或者其他数据源的连接；
- 向数据库发送 SQL 命令；
- 处理数据库的返回结果。

23.1.2　JDBC 的 4 种驱动程序

为了与某个数据库连接，必须要具有适合该数据库的驱动程序。JDBC 驱动程序主要有以下 4 种基本类型。

1．JDBC-ODBC 桥加 ODBC 驱动程序

JDBC-ODBC 桥产品经由 ODBC 驱动程序供 JDBC 访问数据库。ODBC（开放式数据库连接）的出现要比 JDBC 早，广泛地应用于连接各种环境中的数据库。JDBC-ODBC 桥加 ODBC 驱动程序实际是把所有 JDBC 的调用传递给 ODBC，再由 ODBC 调用本地数据库驱动代码。在本书中，将采用此驱动程序+Access 数据库进行演示。这种驱动程序的主要优点有三个：

- 它提供了连接几乎所有平台上的所有数据库的能力。
- 它可能是访问低端桌面数据库（例如 Access）和应用的程序的唯一方式。
- 操作简单，适应初学者练习。

这种驱动程序的缺点如下。

- ODBC 驱动程序需要安装并加载到目标机器上。
- JDBC 和 ODBC 之间的转换将在很大程度上影响系统的性能。

2．本地 API 和部分 Java 编写的驱动程序

这种类型的驱动程序是把客户机 API 上的 JDBC 调用转换为对数据库的调用。也就是说，这种类型的驱动程序是通过调用本地 API 来实现与数据库的通信。这种类型的驱动程序要比采用类型 1 方式的速度快很多，但它仍然存在着一些缺点：

- 它仍然需要在目标机器上安装本地代码。
- JDBC所依赖的本地接口在不同的Java虚拟机供应商及不同的操作系统上是不同的。

3．JDBC 网络纯 Java 驱动程序

这种驱动程序根据三层结构建立：网络协议驱动、中间件服务器和数据库服务。该类型的驱动程序由中间件服务器提供，这样客户端程序可以使用数据库无关的协议和中间件服务器进行通信，中间件服务器再将客户端的 JDBC 调用转发给数据库进行处理。这种方



式的驱动程序有以下优点：

- 不需要客户机上有任何本地代码。
- 不需要客户安装任何程序。
- 大部分功能实现都在 server 端，所以这种驱动可以设计得很小，可以非常快速地加载到内存中。

缺点为：中间件层仍然需要配置其他数据库驱动程序，并且由于多了一个中间层传递数据，它的执行效率还不是最好。

4．本地协议纯 Java 驱动程序

该类型的驱动程序中包含了特定数据库的访问协议，使得客户端可以直接和数据库进行通信。这种方式的驱动程序有以下优点：

- 效率高，速度快。
- 驱动程序可以被动态地下载。

缺点为：对于不同的数据库需要下载不同的驱动程序。

这 4 种驱动程序的属性如表 23.1 所示。

表 23.1　驱动程序的类型及其属性

驱动程序种类	纯 Java	网络协议
JDBC-ODBC 桥加 ODBC	非	直接
本地 API 和部分 Java	非	直接
JDBC 网络纯 Java	是	要求连接器
本地协议纯 Java	是	直接

23.1.3　如何获得 JDBC 驱动程序

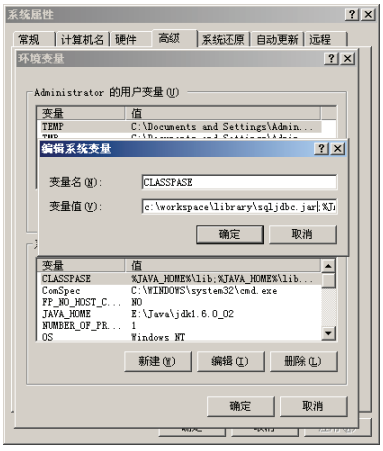


图 23.3　设置驱动程序路径

要想得到某个数据库的驱动程序，可以从该数据库厂商的网站上进行下载，例如 SQL Server 2005 JDBC Driver 的下载地址为：http://www.microsoft.com。也可以访问 Sun 公司的 JDBC 驱动数据库（网址是 http://www.sun.com）得到有关各种数据库的 JDBC 驱动程序的信息。

23.1.4　设置环境变量使得驱动程序可被访问

获得驱动程序以后，需要将驱动程序加入 Classpath 的设定之中。例如作者本人的 SQL Server 2005 的驱动程序路径为“c:\workspace\library\sqljdbc.jar”，则必须在环境变量 Classpath 中添加路径“c:\workspace\library\sqljdbc.jar”，如图 23.3 所示。

23.2　java.sql 包

在 JDBC 介绍中可以看到，JDBC 是采用接口和实现分离的思想设计了 Java 数据库编程的框架。JDBC 的接口包含在 java.sql 及 javax.sql 包中，其中 java.sql 属于 JavaSE，javax.sql

属于 JavaEE。在这里主要对 java.sql 包进行讲解。

java.sql 包中定义了很多接口和类，但是经常使用的却不是很多。在这里介绍几个常用的接口和类。

1．加载驱动程序接口：Driver

java.sql.Driver 是所有 JDBC 驱动程序必须实现的接口。我们在编程中要连接数据库，必须先装载特定厂商提供的数据库驱动程序（Driver），如果使用 JDBC-ODBC 桥驱动，那么装载方法如下。

```
Class.forName="sun.jdbc.odbc.JdbcOdbcDriver";
```

2．管理驱动程序类：DriverManager

DriverManager 类是 JDBC 的管理层，作用于用户和驱动程序之间。它跟踪可用的驱动程序，并在数据库和驱动程序之间建立连接。当 DriverManager 激发 getConnection()方法时，DriverManager 类首先从它已加载的驱动池中找到一个可以接受该数据 URL 的驱动程序，然后请求该驱动程序使用相关的数据库 URL 连接到数据库中。于是，getConnection()方法建立了与数据库的连接。

3．数据库连接接口：Connection

Connection 对象代表与数据库的连接，也就是在已经加载的 Driver 和数据库之间建立连接。你必须创建一个 Connection class 的实例，其中包括你的数据库的信息。

连接过程包括所执行的 SQL 语句和在该连接上所返回的结果。一个应用程序可与单个数据库有一个或多个连接，或者可与许多数据库有连接。DriverManager 的 getConnection()方法，将建立在 JDBC URL 中定义的数据库的 Connection 连接上，代码如下。

```
Connection con = DriverManager.getConnection(url, login,password);
```

4．SQL 声明接口：Statement

java.sql.Statement 提供在基层连接上运行 SQL 语句，并且访问结果。Connection 接口提供了生成 Statement 的方法。在一般情况下，我们通过 connection.createStatement()方法就可以得到 Statement 的实例。

有三种 Statement 对象，它们都作为在给定连接上执行 SQL 语句的包容器：Statement、PreparedStatement（它从 Statement 继承而来）和 CallableStatement（它从 PreparedStatement 继承而来）。

该接口提供了可以被执行的基本的 SQL 语句，作为提高性能的一项措施，PreparedStatement 对象提供了可以与查询信息一起预编译的一种语句类型。CallableStatement 对象用来封装数据库中存储过程的执行。

Statement 提供了许多方法，最常用的方法如下。

- execute()运行语句，返回是否有结果集。
- executeQuery()运行查询语句，返回 ReaultSet 对象。
- executeUpdate()运行更新操作，返回更新的行数。
- addBatch()增加批处理语句。
- executeBatch()执行批处理语句。



➢ clearBatch()清除批处理语句。

Statement 对象用于执行不带参数的简单 SQL 语句，它的典型使用如下所示。

```
…
Connection conn = DatabaseConnection.getConnection(); //获得数据库连接对象
Statement stmt = conn.createStatement();        //创建 SQL 语句对象
ResultSet rst = stmt.executeQuery(“select * from sometable”);
                                                  //执行查询，获得查询结果集
…
```

PreparedStatement 对象用于执行带或不带 IN 参数的预编译 SQL 语句，它的典型使用如下所示。

```
…
Connection conn = DatabaseConnection.getConnection(); //获得数据库连接对象
//创建预编译语句对象
PrepareStatement pstmt = conn.prepareStatement("insert into user_info
values(?,?,?,?,?,?,?)";
pstmt.setString(1."hellking");              //设置预编译 SQL 语句中的第一个参数
pstmt.setInt(2,20);                         //设置预编译 SQL 语句中的第二个参数
pstmt.setBoolean(3,true);                   //设置预编译 SQL 语句中的第三个参数
…
pstmt.executeUpdate();                      //执行 SQL 语句
…
```

CallableStatement 对象用于执行对数据库已存储过程的调用，它的典型使用如下所示。

```
…
Connection conn = DatabaseConnection.getConnection(); //获得数据库连接对象
//创建一个可以滚动的只读的 SQL 语句对象
strSQL = "{call demo_account(?,?,?,?,?,?,?)}";
java.sql.CallableStatement.sqlStmt = conn.prepareCall(strSQL);
sqlStmt.setString(1,"userid_3");        //设置调用存储过程 SQL 语句的第一个参数
sqlStmt.setString(2,"userid_1");        //设置调用存储过程 SQL 语句的第二个参数
…
int I = sqlStmt.executeUpdate();        //执行存储过程
…
```

5．查询结果接口：ResultSet

在 Statement 执行 SQL 语句时，有时会返回 ResultSet 结果集。ResultSet 往往包含的是查询的结果集。此接口抽象了运行 select 语句的结果，提供了逐行访问结果的方法，通过它访问结果的不同字段。结果集通常是一张表，其中有查询所返回的列标题及相应的值。

通过 ResultSet.next()方法把当前的指针向下移动一位。最初它位于第一行之前，因此第一次调用 next 将把指针置于第一行上，使它成为当前行。随着每次调用 next()导致指针向下移动，按照从上至下的次序获取 ResultSet 行。通常情况下，可使用如下的代码遍历 ResultSet 对象内的每一行数据：

```
…
try{
    Connection conn = DatabaseConnection.getConnection();
    Statement stmt = conn.createStatement();
    ResultSet rst = stmt.executeQuery("select * from User_Info");
```

```
        while(rs.next()){
            //显示结果集
            ...
        }
    }
```

读取 ResultSet 对象中每一行数据的方法是调用它的 getXXX()方法，XXX 表示 Java 的数据类型。在使用该方法读取数据时，应该为它指定要读取的列。指定读取的列有两种方法：一种方法是使用列的序号，另一种方法是使用列名。例如，结果集的第 2 列的列名为“StuName”，则下面的两行代码是等价的。

```
...
String name = rs.getString(2);
String name = rs.getString("StuName");
...
```

如果 XXX 所代表的数据类型与结果集中对应列的数据类型不匹配，将会发生类型转换。例如，结果集中“StuID”列的数据类型为 int，那么调用 getString（“StuID”）读取 StuID 一列的信息时，将会把结果集中的 int 类型转化为 String 类型。



23.3 访问数据库

下面通过一个例子来演示 Java 数据库应用程序访问数据库的全过程。在此例中采用数据库使用 Access 数据库，驱动程序使用 JDBC-ODBC 加 ODBC 驱动程序。

23.3.1 创建 Access ODBC 数据源

创建 Access ODBC 数据源分为两步：

（1）创建 Access 数据库；

（2）设置数据源。

下面分别介绍创建数据库的操作和设置数据源的操作。

1．创建 Access 数据库

首先在 Access 数据库中创建数据库 student.mdb，该库中有一个学生信息表，表的名字为 stu_info，如图 23.4 所示。



图 23.4　学生信息表

stu_num、name、birthday、math 与 english 分别表示学号、姓名、出生年月、数学成绩与英语成绩，stu_num 为主键。

2．设置数据源

为了同这个数据库建立连接，首先配置一个 ODBC 数据源，操作步骤如下。

① 打开 Windows 中的【控制面板】窗口，双击【管理工具】图标，出现如图 23.5 所示的界面。

② 双击【数据源（ODBC）】图标，弹出【ODBC 数据源管理器】对话框，如图 23.6 所示。



图 23.5　管理工具面板

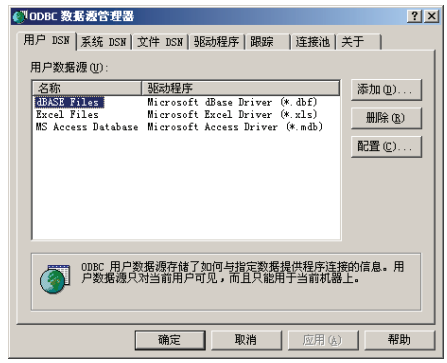


图 23.6　ODBC 数据源管理器

③ 双击【添加】按钮，增加新的数据源，弹出【创建新数据源】对话框，选择【Microsoft Access Driver（*.mdb）】之后，单击【完成】按钮（为数据源选择了驱动程序），如图 23.7 所示。

④ 弹出【ODBC Microsoft Access 安装】对话框，在【数据源名】文本框中为数据源起一个名字，这里起的名字是“student”，如图 23.8 所示。



图 23.7　【创建新数据源】对话框



图 23.8　指定数据源的名称

⑤ 单击【选择】按钮，选择 student.mdb 后单击【确定】按钮，如图 23.9 所示。

⑥ 这时返回【ODBC Microsoft Access 安装】对话框，如果要为数据源 student 设置一个登录名称和密码，就再单击【高级】按钮，否则单击【确定】按钮就完成了数据源设置的全部步骤。现在，给数据源一个登录名称和密码。单击【高级】按钮，打开如图 23.10 所示的【设置高级选项】对话框。

⑦ 在【登录名称】和【密码】文本框中输入用户名和密码，在这里都设为“login”

（这里设置的密码和数据库本身的密码没有关系）。单击【确定】按钮，回到【ODBC Microsoft Access 安装】对话框，再单击【确定】按钮就完成了全部过程。

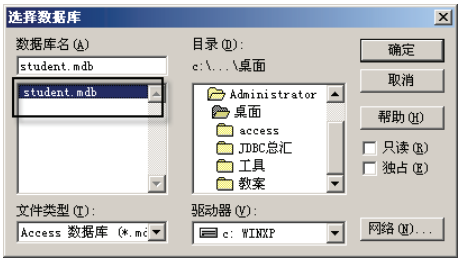


图 23.9 【选择数据库】对话框



图 23.10 【设置高级选项】对话框

23.3.2 创建数据库连接

数据源设置好之后，就可以通过 JDBC 对数据库进行访问了。首先需要在 Java 数据库应用程序中导入 java.sql 包，如下所示。

```
import java.sql.*;                    //导入包
```

在程序中就可以进行数据库连接了。下面的代码将连接数据库的代码写在 TestDB()方法中，使得程序结构更加清晰。

```
public class TestDB {
    private Connection con = null;                  // 数据库的连接
    public TestDB() {
        //桥连接
         try {
             String driver="sun.jdbc.odbc.JdbcOdbcDriver"; //声明桥驱动程序
             String ds = "jdbc:odbc:student";   //student 就是数据源的名字
             String user = "login";             //login 就是数据源的登录名
             String login = "login";            //login 就是数据源的密码
             Class.forName(driver);             //加载数据库的驱动程序
             con = DriverManager.getConnection(ds, user,login);
                                                //连接数据库
             if (con != null) {
                 System.out.println("数据库连接成功！");
             }
        } catch(Exception e) {
            System.out.println("数据库连接失败！"+e.toString());
        }
    }
}
```

上面的程序中，通过“Class.forName(driver);”语句加载了 Access 数据的驱动程序，通过“private Connection con=null;”与“con=DriverManager.getConnection(ds,user,login);”语句，可以连接上 student 数据库。



下面测试一下，看数据库是否连接上。代码如下。

```
public static void main(String[] args) {
    TestDB td = new TestDB();
}
```

若输出结果为“数据库连接成功”，则表明可以对 student 数据库进行操作了。若为“数据库连接失败！+错误提示信息”，则表明连接有错误。仔细检查，确保数据源按上一节的操作进行了设置。

23.3.3　对数据库进行查询

和数据库（数据源）建立连接以后，就可以对数据库进行各种操作了。就好比打电话前，拨通了电话建立了连接以后，就可以通话了一样。对数据库最经常进行的操作，就是数据查询。

在使用 SQL 语句对数据库进行操作之前，需要创建 SQL 声明对象与结果集对象。在“private Connection con=null;”语句后加上如下代码。例如：

```
private Statement st = null;        // 执行 SQL 语句的对象
private ResultSet rs = null;        // 结果集对象
```

通过 Connection 对象的 createStatement()方法可以得到 Statement 的实例，然后通过 Statement 的 executeQuery()方法进行查询，将查询的结果放到 ResultSet 结果集中，最后输出查出的数据。代码如下。

```
//普通查询
public void query(String sql) {
    try {
        st = con.createStatement();     //得到 Statement 的实例
        rs = st.executeQuery(sql);      //执行 SQL 语句,返回结果集
        while (rs != null && rs.next()) {
                            //当返回的结果集不为空时，并且还有记录时
            int stu_num=rs.getInt(1);   //获得当前记录的第 1 个字段的值
            String name = rs.getString("name");
                            //获得当前记录中“name”字段的值
            String birthday=rs.getString("birthday");
                            //获得当前记录中“birthday”字段的值
            int math = rs.getInt("math");//获得当前记录中“math”字段的值
            int english = rs.getInt("english");
                            //获得当前记录中“english”字段的值
            System.out.println("学号="+stu_num+ "\t 姓名="
                            + name+"\t 出生年月="+birthday
                            +"\t 数学成绩="+math+"\t 英语成绩="+english);
        }
    } catch (Exception e) {
        System.out.println("查询数据时出错" + e.toString());
    }
}
```

下面进行测试。在 main()方法中写上如下代码：

```
String name = "陈强";
String sql = "select * from stu_info where name = '"+name+"'";
```

```
td.query(sql);
```

最后输出结果如图 23.11 所示。



图 23.11　查询输出结果 1

23.3.4　增加、修改、删除数据库中数据

想要对数据库进行增加、修改或删除数据的操作，可以通过 Statement 的 executeUpdate()方法，如果在 executeUpdate()方法中的 SQL 语句为 insert 语句，即可向数据库中增加数据；如果在 executeUpdate()方法中的 SQL 语句为 update 语句，则更改数据库中的数据。代码如下所示。

```
//添加、删除、更新
public void add_update_Del(String sql) {
    try {
        st = con.createStatement();              // 生成 Statement 对象
        int x = st.executeUpdate(sql);           // 执行操作
        System.out.println("操作成功" + x);
    } catch (Exception e) {
        System.out.println("数据修改时有误" + e.toString());
    }
}
```

下面向数据库添加一条记录，代码如下。

```
String sql = "insert into stu_info values(7,'宋强','86-9-16',89,84)";
td.add_update_Del(sql);
```

这时使用 SQL 语句进行查询，代码如下。

```
String name = "宋强";
String sql = "select * from stu_info where name = '"+name+"'";
td.query(sql);
```

最后输出结果如图 23.12 所示。



图 23.12　查询输出结果 2

若想更改表中的数据，可以使用如下语句：

```
String sql = "update stu_info set math = 95 where name = '宋强'";
td.add_update_Del(sql);
```

这时就将宋强的数学成绩更改为 95。若要删除一条记录，可以使用如下语句：

```
String sql = "delete from  stu_info where name = '宋强'";
td.add_update_Del(sql);
```

最后在数据库看到的表 stu_info 如图 23.13 所示。

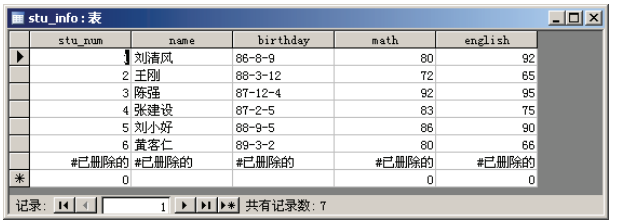


图 23.13　stu_info 表

23.3.5　使用 PreparedStatement 访问数据库

PreparedStatement 接口继承 Statement 类，并与之在两方面有所不同。

- PreparedStatement 实例包含已编译的 SQL 语句。由于 PreparedStatement 对象已预编译过，所以其执行速度要快于 Statement 对象。因此，多次执行的 SQL 语句经常创建 PreparedStatement 对象，以提高效率。
- 包含于 PreparedStatement 对象中的 SQL 语句可具有一个或多个 IN 参数。IN 参数的值在 SQL 语句创建时未被指定。相反，该语句为每个 IN 参数保留一个问号（“?”）作为占位符。每个问号的值必须在语句执行之前，通过适当的 SetXXX 方法来提供。

例如在上一节的学生信息表中，需要反复插入学生的信息记录。这时可以先创建一个预编译的方法。代码如下。

```
    // 预编译插入操作
    public void preparedAdd(int num, String name, String birthday, int math,
int english) {
        try {
            String sql = "insert into stu_info (stu_num,name,birthday,math,
english) values(?,?,?,?,?)";
            pst = con.prepareStatement(sql);// 生成执行 SQL 语句的预编译语句对象
            pst.setInt(1, num);              //设置预编译语句对象的第一个参数
            pst.setString(2, name);          //设置预编译语句对象的第二个参数
            pst.setString(3, birthday);      //设置预编译语句对象的第三个参数
            pst.setInt(4, math);             //设置预编译语句对象的第四个参数
            pst.setInt(5, english);          //设置预编译语句对象的第五个参数
            int x = pst.executeUpdate();     //执行预编译语句对象语句
            System.out.println("操作成功" + x);
        } catch (Exception e) {
            System.out.println("预编译操作不成功！" + e.toString());
        }
    }
```

最后调用此方法来插入数据。代码如下。

```
td.preparedAdd(7,"宋强","86-9-16",89,84);
```

这样就使用 PreparedStatement 对数据库进行了操作。

23.4　事务

在 Connection 对象上可以使用事务（Transaction）。用户可以把一系列对数据库的改变

动作作为一个整体来对待，这个整体就是一个事务。

例如，当用户打算将一笔账转移到另一笔账上时，用户从一笔账上减去一部分，则必须在另外一笔账上增加这部分。如果任何一次增减动作失败，都将使用户的账单不再平衡。如果将这些增减动作放在一起处理，可以确保所有的增减操作或者全部执行，或者全部不执行，如果所有的改变都正确，则使用 commit 将这些改变动作全部存入数据库，否则就使用 rollback 取消所有的改变动作，而这时数据库中的数据和执行事务前的数据是相同的。

默认情况下新产生的连接为自动提交模式（auto-commit），即每一个 SQL 语句是独立的事务，执行完成 SQL 语句就立即提交了。Connection 对象有一个 setAutoCommit(false)，使用它则自动提交模式关闭了。这让用户能够自己组合 SQL 语句，正常执行完成所有的 SQL 语句后再调用 commit()提交事务。若其中某个 SQL 语句执行失败，则必须调用 rollback()取消这次事务。

例如在本章的学生信息表中，如果想将学号为 2 和 3 的同学的数学成绩都加上 10 分，但是学生信息中有数学成绩的最高分不得超过 100 分的规定，这时可以使用事务，若加上 10 分后都没超过 100 分，则提交，否则回滚操作。代码如下所示。

```
public void transaction(){
    try {
        con.setAutoCommit(false);           // 关闭自动提交事务模式
        st = con.createStatement();         // 创建 SQL 语句对象
        //以下四句为查询学号为 2 的学生的数学成绩,并将其加 10
        String sql1 = "select math from stu_info where stu_num = 2";
        rs = st.executeQuery(sql1);//执行查询语句，并将查询结果返回给结果集 rs
        rs.next();                  //获得结果集中下一条记录
        int math1= rs.getInt(1)+10;
        //获得当前记录中第一列的值，加 10 后赋给变量 math1
        System.out.println(math1);          //输出改变以后的 math1 的值
        //以下四句为查询学号为 3 的学生的数学成绩,并将其加 10
        String sql2 = "select math from stu_info where stu_num = 3";
        rs = st.executeQuery(sql2);//执行查询语句，并将查询结果返回给结果集 rs
        rs.next();                  //获得结果集中下一条记录
        int math2 = rs.getInt("math")+10;
        //获得当前记录中第一列的值，加 10 后赋给变量 math1
        System.out.println(math2); //输出改变以后的 math1 的值
        //更改两名学生的数学成绩
        String sql3 = "update stu_info set math = "+math1+" where stu_num = 2";
        st.executeUpdate(sql3);     // 执行操作
        String sql4 = "update stu_info set math = "+math2+" where stu_num = 3";
        st.executeUpdate(sql4);     // 执行操作
        con.commit();                       // 提交事务
        System.out.println("提交成功");
    } catch (SQLException e) {              //捕获数据库操作异常
        try{
            con.rollback();                 //如果有异常发生，就回滚事务
            System.out.println("事务回滚");
        }catch(Exception ignored){}
    }
}
```

因此 math2 的值超过 100，所以事务回滚。执行结果如图 23.14 所示。



图 23.14　事务回滚

这时如果将加的分数改为 5，其输出结果如图 23.15 所示。



图 23.15　事务提交成功

在数据库中可以看到，stu_info 表如图 23.16 所示。



图 23.16　stu_info 表

最后当对数据库访问结束后，需要关闭与数据库的连接。代码如下所示。

```
// 关闭数据库连接
public void close() {
    try {
        if (con != null) {                    //如果连接对象不为空
            con.close();                      //调用 colse()方法关闭连接
        }
    } catch (Exception e) {
        System.out.println("关闭数据库时出现异常");
    }
}
```

23.5　小结

在本章中，介绍了 Java 的 JDBC 技术，以及如何使用 java.sql 包中的相关类编写数据库应用程序，最后还介绍了事务管理。其中应用 java.sql 包中的类编写数据库应用程序应是读者本章学习的重点。而在数据库程序中处理事务是本章学习的难点，读者应结合本书所提供的示例上机调试运行，以帮助理解相关的概念。下一章将介绍如何使用 Swing 组件开发实用的数据库应用程序。

第 24 章　使用 Swing 组件创建数据库应用程序

在上一章中，通过JDBC对数据库进行操作，并将结果显示在控制台中，本章通过Swing组件显示数据库中的信息。主要内容包括：

- 使用 JComboBox 组件创建数据库应用程序。
- 使用 JList 组件创建数据库应用程序。
- 使用 JTable 组件创建数据库应用程序。

24.1　使用 JComboBox 组件创建数据库应用程序

组合框控件（JComboBox）是桌面应用程序中经常使用到的一个控件。它可以存放数量较多的选项，并且占用较小的桌面空间。在实际应用中，经常使用 JComboBox 控件来显示从数据库中查询的批量数据。本节将学习如何使用 JComboBox 控件创建数据库应用程序。

24.1.1　创建一个 JComboBox

使用下拉式列表组件（JComboBox）可以制作一个弹出式的数据项选择列表，让用户在一系列的选项中选出需要的值。图 24.1 为 JTable 类在继承树中的层次。

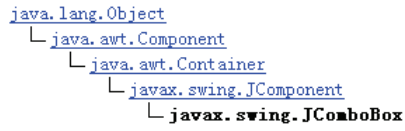


图 24.1　JTable 类在继承树中的层次

表 24.1 列出了该类常用的方法。

表 24.1　JComboBox 类常用方法

方　　法	方法说明
JComboBox()	构造方法，以系统默认的 ComboBox Model 模型创建一个新的下拉式列表组件对象
JComboBox(ListModel dateModel)	构造方法，使用指定的模型(ComboBox Model)构造一个下拉式列表
addActionListerner(ActionEvent I)	将动作监听器对象 I 加入本组件的监听对象中
addItem(Object obj)	为下拉列表增加数据项
getItemAt(int index)	返回指定索引位置的列表项



续表

方　法	方法说明
getItemCount()	返回列表中的列表项个数
getSelectedItem()	返回当前选择的项
getSelectedIndex()	返回当前选择项的索引位置

下面创建一个显示学生信息的组合框。在此例中，仍然使用上一章的 stu_info 表。

【例 24.1】创建一个组合框，组合框中的数据项的值为学生的姓名，学生的姓名是通过 JDBC 从数据库查询出来的。

（1）编写一个 TestDB 类，在该类中定义了 query()方法，用来进行姓名查询，并返回查询的结果集对象。代码如下所示。

```
import java.sql.*;
public class TestDB {
    private Connection con = null;                  // 声明数据库连接对象
    private Statement st = null;                    // 声明 SQL 语句对象
    private ResultSet rs = null;                    // 声明结果集对象
    public TestDB() {
        // 桥连接
        try {
            String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
                                                    // 声明桥驱动程序
            String ds = "jdbc:odbc:student";        // student 就是数据源的名字
            String user = "login";                  // 设置用户名
            String login = "login";                 // 设置密码
            Class.forName(driver);                  // 加载数据库的驱动程序
            con = DriverManager.getConnection(ds, user, login);
                                                    // 获得与数据库的连接
            if (con != null) {                      //如果数据库连接成功，输出信息
                System.out.println("数据库连接成功！");
            }
        } catch (Exception e) {
            System.out.println("数据库连接失败！" + e.toString());
        }
    }
    // 普通查询
    public ResultSet query(String sql) {
        try {
            //结果集类型设为 TYPE_SCROLL_SENSITIVE 并发类型设为 CONCUR_READ_ONLY
            st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,

ResultSet.CONCUR_READ_ONLY);
            rs = st.executeQuery(sql);      // 执行 SQL 语句,返回结果集
        } catch (Exception e) {
            System.out.println("查询数据时出错" + e.toString());
        }
        return rs;                          //返回查询结果集
    }
    // 关闭数据库连接
    public void close() {
        try {
```

```
            if (con != null) {
                con.close();
            }
        } catch (Exception e) {
            System.out.println("关闭数据库时出现异常");
        }
    }
}
```

在此类中，将查询方法的返回值类型设为 ResultSet，结果集类型设为 TYPE_SCROLL_SENSITIVE，表明结果集可滚动，并且受 ResultSet 底层数据的更改而影响 ResultSet 对象的类型。并发类型设为 CONCUR_READ_ONLY，代表这是不可以更新的 ResultSet 对象的并发模式。

（2）创建一个组合框（JComboBox）来将 stu_info 表中的姓名显示出来。代码如下所示。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;
import java.util.Vector;
public class TestJComboBox{
    public static void main(String[] args){
        JFrame f=new JFrame("姓名");     //创建底层容器 JFrame
        Container contentPane=f.getContentPane();
                                         //通过 getgetContentPane 创建内容面板
        contentPane.setLayout(new FlowLayout());
                                            //设置内容面板的布局方式为流式布局
        String str="select name from stu_info";         //查询语句
        TestDB db=new TestDB();             //创建 TesbDB 实例，连接数据库
        ResultSet rs=db.query(str);         //调用 query 方法进行查询
        // 以下代码为将结果集写入数组 name 中/
        try {
            rs.last();                      //移动到结果集的最后一行
            int I = rs.getRow();            //得到行数，赋给变量 i
            rs.beforeFirst();               //移回结果集首行
            String name[] = new String[i]; //创建数组 name，长度为 i
            int m = 0;
            while(rs.next()){               //循环取出数据，赋给数组 name
                name[m] = rs.getString(1);
                m++;
            }
            //创建下拉列表实例，列表项的值为数组 name
            JComboBox combo1 = new JComboBox(name);
            contentPane.add(combo1);        //将下拉列表放入内容面板中
        }catch (SQLException e) {
            e.printStackTrace();
        }
        f.setSize(150,200);                 //设置容器宽为 150，高为 200
        f.setVisible(true);                 //设置容器可见
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                                            //设置监听器，关闭窗口后，程序结束
```



```
                System.exit(0);
            }
        });
    }
}
```

在此例中，首先创建底层容器 JFrame，虽然 JFrame 是底层框架容器，但并不能直接将组件加入其中，所以使用 getgetContentPane()获取 JFrame 的内容面板。接着创建查询语句，调用刚才写好的 TestDB 类，对数据库进行访问。下面循环取出得到数据，并将其放入数组中，然后创建组合框，组合框列表项的值为数组 name，这时再通过内容面板的 add()方法加入下拉列表，最后设置 JFrame 窗口的宽和高，并将其显现出来。程序的运行结果如图 24.2 所示。



图 24.2　姓名下拉列表

24.1.2　利用 DefaultComboBoxModel 创建 JComboBox

java 对于 JComboBox 提供了一个 DefaultComboBoxModel 实体类。此类继承了 AbstractListModel 抽象类，利用 DefaultComboBoxModel 这个类我们可以很方便地做到动态更改 JComboBox 的项目值。下面通过 DefaultComboBoxModel 来添加一个列表

【例 24.2】创建一个组合框，组合框中的数据项的值为学生的姓名，学生的姓名是通过 JDBC 从数据库查询出来的，并使用 DefaultComboBoxModel 动态填充组合框信息。

```
import java.awt.*;
import java.awt.event.*;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.swing.*;
public class TestDefaultComboBoxModel {
    public static void main(String[] args){
        JFrame f = new JFrame("姓名");                    //创建窗体对象 f
        Container contentPane = f.getContentPane();//获得窗体 f 的内容面板
        contentPane.setLayout(new FlowLayout());
                                                  //设置内容面板的布局方式为流式布局
        String str="select name from stu_info";     //查询语句
        TestDB db=new TestDB();                   //创建 TesbDB 实例，连接数据库
        ResultSet rs=db.query(str);                       //调用 query 方法进行查询
        // 以下代码将结果集写入数组 name 中
        try {
            rs.last();                            //移动到结果集的最后一行
            int i = rs.getRow();                  //得到行数，赋给变量 i
            rs.beforeFirst();                     //移回结果集首行
            String name[] = new String[i];  //创建数组 name，长度为 i
            int m = 0;
            while(rs.next()){                     //循环取出数据，赋给数组 name
                name[m] = rs.getString(1);
                m++;
            }
            DefaultComboBoxModel mode=new DefaultComboBoxModel();
                                                  //创建下拉列表模型
```

```
            for (int k=0;k<name.length;k++){
                mode.addElement(name[k]);                    //添加列表项
            }
            JComboBox combo=new JComboBox(mode);             //创建下拉列表
            contentPane.add(combo);                          //添加下拉列表
        }catch (SQLException e) {
            e.printStackTrace();
        }
        f.setSize(150,200);                                  //设置窗体的大小
        f.setVisible(true);                                  //设置窗体可见
        f.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                                                  //设置监听器，关闭窗口后，程序结束
                    System.exit(0);
                }
        });
    }
}
```

在此例中，首先创建一个组合框模型对象，然后通过 addElement()方法加入列表项，使用这个方法可以动态更改 JComboBox 的项目值，最后使用组合框模型对象创建下拉列表。程序运行结果与图 24.2 一致。

24.2 使用 JList 组件创建数据库应用程序

列表控件（JList）是另一种常用的选择控件，支持单选和多选，还支持带图标的选择项。在实际应用中，经常使用 JList 控件来显示从数据库中查询的批量数据。本节介绍如何使用 JList 组件创建数据库应用程序。

24.2.1 利用 DefaultListModel 创建 JList

列表与组合框十分相似，不同之处是用户始终可以看到列表的内容。列表一般有较多的列表项，占据空间也大，因此通常将其放置在滚动面板上，这样就可以提供一个文本数据项可以卷动的列表。JList 类在继承树中的层次如图 24.3 所示。

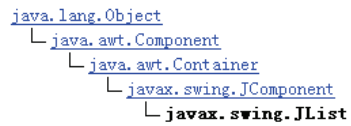


图 24.3 JTable 类在继承树中的层次

表 24.2 列出了 Jlist 类常用方法。

表 24.2 JList 类常用方法

方　法	方法说明
JList()	构造方法，创建一个新的列表对象
JList(ListModel dateModel)	构造方法，使用列表模型(ListModel)建立一个新的列表
JList(Array ListData)	构造方法，使用数组对象创建一个新的组件对象
JList(Vector listData)	构造方法，使用 Vector 对象创建一个新的组件对象
void setListData(Array listData)	方法，为列表设置为数组列表项
void setListData(Vector listData)	方法，为列表设置为数组列表项

下面使用 DefaultListModel 来创建一个列表（JList），同样也是将 stu_info 表中的 name



取出来，使用 JList 进行显示。

【例 24.3】创建一个列表，列表中的数据项的值为学生的姓名，学生的姓名是通过 JDBC 从数据库查询出来的。代码如下所示。

```
import java.awt.*;
import java.awt.event.*;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.swing.*;
public class TestDefaultListModel {
    public static void main(String[] args){
         JFrame f=new JFrame("姓名");                    //创建窗体对象
         Container contentPane=f.getContentPane(); //获得窗体对象的内容面板
         String str="select name from stu_info";   //查询语句
          TestDB db=new TestDB();                  //创建 TestDB 实例，连接数据库
          ResultSet rs=db.query(str);              //调用 query 方法进行查询
         // 以下代码为将结果集写入数组 name 中
         try {
            rs.last();                             //移动到结果集的最后一行
            int I = rs.getRow();                   //得到行数，赋给变量 i
            rs.beforeFirst();                      //移回结果集首行
            String name[] = new String[i];         //创建数组 name，长度为 i
            int m = 0;
            while(rs.next()){                      //循环取出数据，赋给数组 name
                name[m] = rs.getString(1);
                m++;
            }
            DefaultListModel mode = new DefaultListModel();
                                                   //创建列表模型
            for (int k = 0;k<name.length;k++){ //循环获取名字
                mode.addElement(name[k]);          //添加列表项
            }
            JList  list = new JList (mode);        //创建列表
            list.setVisibleRowCount(3);            //设置列表首选大小为 3
            contentPane.add(new JScrollPane(list));//添加列表至可滚动的视图中
         }catch (SQLException e) {
            e.printStackTrace();
         }
         f.pack();          //pack()方法用来调整窗口的大小，以适合 list 的首选大小
         f.setVisible(true);
         f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                           //设置监听器，关闭窗口后，程序结束
                System.exit(0);
            }
         });
    }
}
```

在此例中，首先使用 DefaultListModel 创建列表模型，然后通过 addElement()方法添加列表项，接着通过列表模型创建列表，下面通过列表的 setVisibleRowCount()的方法设置列表初始可见长度为 3，并将 list 放入 JScrollPane 视图中，当 list 的内容超过视图大小时，

就会产生滚动条。最后调用 JFrame 容器的 pack()方法，让其调整窗口的大小，以适合其 list 的首选大小。其效果如图 24.4 所示。



图 24.4　姓名列表

24.2.2　利用 ListModel 与 AbstractListModel 创建 JList

ListModel 是一个接口，主要的功能是定义一些方法，让 JList 或 JComboBox 这些组件取得每个项目的值，并可限定项目的显示时机与方式，表 24.3 列出了 ListModel 这个 interface 所定义的方法。

表 24.3　ListModel 的方法

void addListDataListener(ListDataListener l)	当 data model 的长度或内容值有任何改变时，利用此方法就可以处理 ListDataListener 的事件。data model 是 vector 或 array 的数据类型，里面存放 List 中的值
Object getElementAt(int index)	返回在 index 位置的 Item 值
int getSize()	返回 List 的长度
void removeListDataListener(ListDataListener l)	删除 ListDataListener

刚才介绍过 JList 有这样一个构造器：JList(ListModel dataModel)，因此必须实现 ListModel 所有的方法，才能利用上面这个构造器建立 JList。Java 提供了 AbstractListModel 这个抽象类，此抽象类实现了 addListDataListener()与 removeListDataListener()这两个方法。若直接继承 AbstractListModel 类，就不需实现这两个方法，只需要实现 getElementAt()与 getSize()方法即可。下面通过一个例子来利用 ListModel 与 AbstractListModel 创建 JList。

【例 24.4】创建一个列表，列表中的数据项的值为学生的姓名，学生的姓名是通过 JDBC 从数据库查询出来的。实现步骤如下。

① 在 TestDB 类中添加一个取得姓名的方法 getData()，代码如下所示。

```
public String[] getData()  {
    ResultSet rs = query("select name  from stu_info");
    try {
        rs.last();                          //移动到结果集的最后一行
        int i = rs.getRow();                //得到行数，赋给变量 i
        rs.beforeFirst();                   //移回结果集首行
        String name[] = new String[i];      //创建数组 name，长度为 i
        int m=0;
        while(rs.next()){                   //循环取出数据，赋给数组 name
            name[m] = rs.getString(1);
            m++;
        }
        return name;
    }catch (Exception e) {}
    return null;
}
```

② 编写一个 TestAbstractListModel 来显示姓名，代码如下所示。

```
import java.awt.*;
import java.awt.event.*;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.swing.*;
```

```
public class TestAbstractListModel {
    public TestAbstractListModel() {
        JFrame f = new JFrame("JList");
        Container contentPane = f.getContentPane();
        ListModel mode = new DataModel();       // 创建 ListModel 对象
        JList list = new JList(mode);  // 利用 ListModel 建立一个 JList.
        list.setVisibleRowCount(5);// 设置程序一打开时所能看到的数据项个数
        contentPane.add(new JScrollPane(list));// 向内容面板中添加 JList 组件
        f.pack();
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
    public static void main(String[] args) {
        new TestAbstractListModel();
    }
    // 创建继承自 AbstractListModel 抽象类的内部类 DataModel
    class DataModel extends AbstractListModel {
        TestDB db= new TestDB();
        //getxElementAt()方法中的参数 index,系统会自动由 0 开始计算,不过要自己做
累加的操作
        public Object getElementAt(int index) {
    // 实现继承的 getElementAt()方法
            return (index + 1) + "." + db.getData()[index++];
        }
        public int getSize() {              // 实现继承的 getSize()方法
            return db.getData().length;
        }
    }
}
```

在上面的程序中，由于在 DataMode 类中继承 AbstractListModel，并实现了 getElementAt()与 getSize()方法，因此可以由 DataModel 类产生一个 ListModel 的实体来。

当程序调用 setVisible(true)方法时，系统会先自动调用 getSize()方法，看看这个 list 长度有多少；然后再调用 setVisibleRowCount()方法，看要一次输出多少数据；最后调用 getElementAt()方法，将 list 中的项目值填入 list 中。

在 AbstractListModel 中，首先通过 TestDB 的 getData()得到姓名数组，然后实现 getElementAt()与 getSize()方法。程序运行结果如图 24.4 所示。

24.3　使用 JTable 组件创建数据库应用程序

JTable 是 Swing 中最复杂的控件之一，但是在桌面程序开发中，表又是最具表现力的形式之一，因此熟练使用 JTable 进行数据库应用程序的编写，是对进行桌面程序开发的程序员的要求之一。本节介绍 JTable 的常见用法。

24.3.1 与 JTable 相关的类

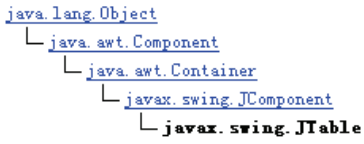


图 24.5 JTable 类在继承树中的层次

JTable 组件的父类依然是 JComponent，图 24.5 显示了 JTable 类在继承树中的层次。

使用表格类（javax.swing.JTable）的构造方法 JTable(TableMode model)可创建指定表格模型的表格对象，参数 model 是实现 java.swing.table.TreeNode 接口的表格模型。表格类 JTable 的常用方法如表 24.4 所示。

表 24.4 表格类 JTable 的常用方法

JTable()	构造一个默认的 JTable，使用默认的数据模型、默认的列模型和默认的选择模型对其进行初始化
JTable(intnumRows,int numColumns)	使用 DefaultTableModel 构造具有 numRows 行和 numColumns 列个空单元格的 JTable
JTable(Object[][]rowData,Object[] columnNames)	构造一个 JTable 来显示二维数组 rowData 中的值，其列名称为 columnNames
int getColumnCount()	返回表格的列数
int getRowCount()	返回表格的行数
void setValueAt(int row,int column)	设置表格行索引值为 row、列索引值为 column 的单元格的内容
void getValueAt(int row,int column)	得到表格行索引值为 row、列索引值为 column 的单元格的内容
int getRowCount()	返回表格列索引值为 column 的列名称

下面介绍几个与 JTable 相关的类。

1．TableModel

TableModel 类本身是一个接口，在这个接口里面定义了若干的方法：包括了存取表格字段（cell）的内容、计算表格的列数等的基本存取操作，让设计者可以简单地利用 TableModel 来实现他所想要的表格。表 24.5 列出了 TableModel 的常用方法。

表 24.5 TableModel 的常用方法

void addTableModelListener(TableModelListener l)	使表格具有处理 TableModelEvent 的能力。当表格的 TableModel 有所变化时，会发出 TableModel Event 事件信息
Class getColumnClass(int columnIndex)	返回字段数据类型的类名称
int getColumnCount()	返回字段(行)数量
String getColumnName(int columnIndex)	返回字段名称
int getRowCount()	返回数据列数量
Object getValueAt(int rowIndex,int columnIndex)	返回数据某个 cell 中的值
boolean isCellEditable(int rowIndex,int columnIndex)	返回 cell 是否可编辑,true 的话为可编辑
void removeTableModelListener(TableModelListener l)	从 TableModelListener 中移除一个 listener
void setValueAt(Object aValue,int rowIndex,int columnIndex)	设置某个 cell(rowIndex,columnIndex)的值

由于 TableModel 本身是一个接口，因此直接实现此界面来建立表格并不容易，不过 java 提供了两个类分别实现了这个界面，一个是 AbstractTableModel 抽象类，一个是 DefaultTableModel 实体类。前者实现了大部分的 TableModel 方法，让用户可以很有弹性地构造自己的表格模式；后者继承前者类，是 Java 默认的表格模式。这三者的关系如下所示：



TableModel---implements--->AbstractTableModel-----extends--->DefaultTableModel

2．AbstractTableModel

java 提供的 AbstractTableModel 是一个抽象类，这个类帮我们实现大部分的 TableModel 方法，除了 getRowCount()，getColumnCount()，getValueAt()这三个方法外。因此我们的主要任务就是去实现这三个方法。利用这个抽象类就可以设计出不同格式的表格。

3．TableColumnModel

TableColumnModel 本身是一个接口，里面定义了许多与表格的“列（行）”有关的方法，例如增加列，删除列，设置与取得“列”的相关信息。通常不会直接实现 TableColumnModel 界面，而是会利用 JTable 的 getColumnModel() 方法取得 TableColumnModel 对象，再利用此对象对字段做设置。举例来说，如果想设计的表格是包括有下拉式列表的组合框，就能利用 TableColumnModel 来达到这样的效果。

24.3.2　利用 DefaultTableModel 创建 JTable

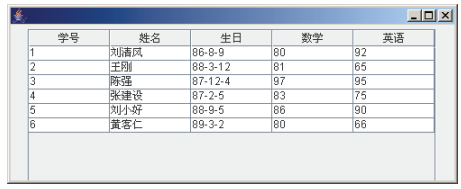
DefaultTableMode 类继承 AbstractTableModel 抽象类而来，且实现了 getColumnCount()、getRowCount() 与 getValueAt() 三个方法。因此在实际使用上，DefaultTableModel 比 AbstractTableModel 要来得简单许多。在 DefaultTableMode 中，常常使用 Vector（向量）来填充表格中的数据。

【例 24.5】使用 DefaultTableModel 来创建一个表格（JTable 对象），表格的内容为 stu_info 表中的学生信息。代码如下所示。

```
import java.awt.*;
import java.awt.event.*;
import java.sql.*;
import java.util.Vector;
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
public class TestDefaultTableModel extends JFrame {
    Vector colsv = new Vector();            //创建一个向量
    JTable table;                           //声明一个 JTable 对象
    DefaultTableModel tablemodel;           //创建 DefaultTableModel 模型
    public TestDefaultTableModel() {
        this.setLayout(new FlowLayout());
        // 将 String 对象"学号", "姓名", "生日", "数学"加入向量中用于显示表头
        colsv.add("学号");
        colsv.add("姓名");
        colsv.add("生日");
        colsv.add("数学");
        colsv.add("英语");
        tablemodel = new DefaultTableModel(new Vector(), colsv);   //实例
化表格模型，列值为向量 colsv
        String str = "select * from stu_info";
        TestDB db = new TestDB();
        ResultSet rs = db.query(str);       //执行查询操作
        //  将查询出的结果集中的值放到 vector
        Vector value = new Vector();
```

```
            try {
                while (rs.next()) {                    //循环遍历结果集 rs 中的记录
                    Vector vc = new Vector();          //创建向量对象
                    vc.add(rs.getString(1));
                                    //将从当前记录中获得的相应字段的值添加到向量中
                    vc.add(rs.getString(2));
                    vc.add(rs.getString(3));
                    vc.add(rs.getString(4));
                    vc.add(rs.getString(5));
                    value.add(vc);
                }
                tablemodel.setDataVector(value, colsv);
                                    //在表格中写入新的行，其值为向量 value
            } catch (SQLException e) {
                e.printStackTrace();
            }
            table = new JTable(tablemodel);            //使用 tablemodel 创建表对象
            this.add(new JScrollPane(table));          //将表格添加到滚动面板中
            this.setSize(500, 300);                    //设置窗体大小
            this.setVisible(true);                     //使窗体可见
            // 设置监听器，关闭窗口后，程序结束
            this.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });
        }
        //主程序
        public static void main(String[] args) {
            new TestDefaultTableModel();
        }
    }
```

在此例中，创建 DefaultTableModel 模型，其列值为向量 colsv，等值为向量 value。在 DefaultTableModel 中使用向量来添加表格中数据非常的方便。程序结果如图 24.6 所示。



学号	姓名	生日	数学	英语
1	刘浦凤	86-8-9	80	92
2	王刚	88-3-12	81	65
3	陈强	87-12-4	97	95
4	张建设	87-2-5	83	75
5	刘小妍	88-9-5	86	90
6	黄客仁	89-3-2	80	66

图 24.6　表格 stu_info

24.4　小结

在本章中，通过若干实例学习了如何使用 Swing 组件 JComboBox、JList 和 JTable 创建数据库应用程序。读者在学习本章内容时，重点掌握 JComboBox 组件、JList 组件和 JTable 组件显示数据的方式。而这些组件所采用的 MVC 模式可能是读者学习的一个难点。建议读者将本章的示例上机调试和运行，细心体会 MVC 开发的模式。读者若想进一步深入学习 MVC，可参考相关的书籍。

从下一章开始，将通过实际案例，为读者讲解 Java 各种应用程序开发的流程和技巧。

5

第 5 篇
Java 实战篇

- 第 25 章　案例 1：拼图游戏
- 第 26 章　案例 2：心心网络聊天室
- 第 27 章　案例 3：惠文教育教务管理系统

第25章　案例1：拼图游戏

通过前面的学习，读者已经基本掌握了 Java 程序设计语言的语法基础、面向对象程序设计的概念及 Swing 图形界面程序的编程。本章通过一个拼图游戏程序的开发案例，详细讲解如何综合运用 Swing 组件和布局管理器进行程序界面布局，响应事件处理的方法，以及应用程序的开发思路。

25.1　软件简介

拼图游戏是很常见而且很受欢迎的一种小游戏。所谓拼图游戏，是指将一个完整的图片分割成若干个规则的小图片，然后将这些小图片随机地拼接在一起。玩家按照原图，通过移动各个图片，最后重新拼接出正确的图片。

25.1.1　应用背景

拼图游戏是老少皆宜的一个小游戏。在工作和学习之余，可以自己单独玩，也可以和家人一起玩，不但益智，而且可以达到放松身心的目的。现在计划开发一单机版拼图游戏，使用 Java 语言，可以双击独立运行。

25.1.2　功能介绍

拼图游戏界面的效果如图 25.1 所示。在图 25.1 右边的小图片是供玩家参考的原图。左边带网格的大图片为初始打乱的拼接图，每一个小网格都是一个小图片，在开始游戏时这些小图片是杂乱无章的，如图 25.1 所示，在图中有一个网格是空白的，玩家可以通过单击与其相邻的网格，即位于其上、下、左、右的 4 个网格，将被单击网格中显示的小图片移动到空白网格中，然后被单击的网格将变为空白，经过这样反复地移动，要求最终将图 25.1 左边所示的图片重新拼接成图 25.1 右边所示原始效果图。

单击【开始】|【新开局】命令，会重新开始新的一局，这时会重新对图片进行布局。单击【开始】|【退出】命令，会退出游戏，结束程序，如图 25.2 所示。

单击【关于】|【游戏说明】命令，会弹出对话框，对游戏的操作方法加以简单说明。单击【关于】|【关于作者】命令，会弹出对话框，说明软件的版本号、作者信息等，如图

25.3 所示。

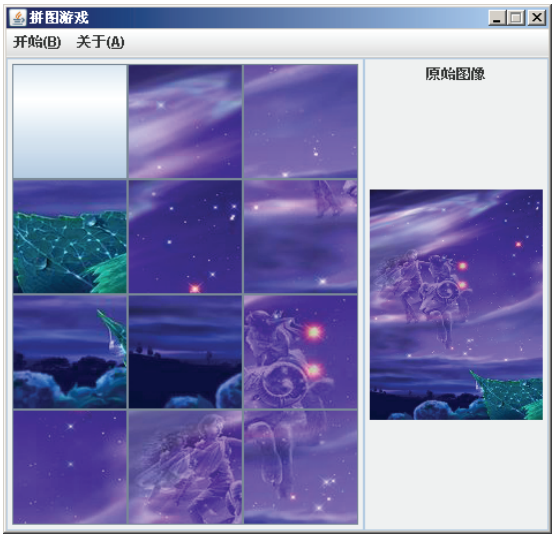


图 25.1　拼图游戏



图 25.2 【开始】菜单



图 25.3　【关于】菜单

25.2　素材准备

拼图游戏所需的素材比较简单，仅为需要拼接的图片，如图 25.1 所示，大小为 400*300（单位像素），拆分的小图片大小为 100*100 像素。对于图 25.1 中被拆分的小图，不需要另外准备，在内存中对图片进行截取和拆分即可。

首先，准备好原始图片，在与源文件相同的目录下创建文件夹 background，然后将原始文件存放于 background 文件夹下。在程序中，使用 javax.imageio.ImageIO 类的 read()方法，将原始图像读入内存，保存为 java.awt.image.BufferedImage 对象。

在内存中，使用 BufferedImage 对象的 getSubimage(int x, int y, int wid, int height)方法，返回位于图像指定位置指定大小的子图像，并保存在 BufferedImage 对象数组中，在构造按钮对象时从数组中读取使用。

对于图 25.1 中右边的原始参考图，是在内存中对原始图片进行缩放以后显示的结果。在程序代码部分会有详细说明。

25.3　算法设计

要实现本游戏上述功能，关键考虑如下几个问题：如何加载和拆分图片、如何实现图片的随机排列、如何在程序中实现图片的移动。这实际上是算法的设计问题。本节简要分析一下拼图游戏的算法设计和实现。

25.3.1　加载和拆分图片

将原始图片放在 background 目录下，使用 javax.imageio.ImageIO 类的 read()方法，加载图片到内存中，保存到缓冲图像对象 BufferedImage 中。代码如下所示。

```
BufferedImage image = null;
```

```
try{
      image = ImageIO.read(new File(filename)); //使用 ImageIO 对象读取图像文件
}catch (Exception e){           //在读取图像文件时，有可能发生异常，要捕获和处理
      System.out.println("加载图像错误！");
}}
```

成功加载图像以后，在内存中对加载的图像进行拆分，使用 BufferedImage 类的 getSubimage()方法。代码如下所示。

```
    ArrayList<BufferedImage> subimage = new ArrayList<BufferedImage>(this.
row*this.col);
    for (int i = 0; i <this.row; i++)
          for (int j = 0; j <this.col; j++)
                //返回由指定矩形区域定义的子图像
                subimage.add((i*this.col + j),image.getSubimage(j*len, i*len,
len,len));
```

在这里，定义了一个 BufferedImage 类型的动态数组 subimage，然后使用双重循环，获取原始图像的各个子图像，并保存到动态数组中 subimage。注意这里定义动态数组使用了泛型。

25.3.2 实现图片的随机排列

在程序一开始，拆分过后的图像应该是随机排列的。随机排列可以有两种方式：一是将图像顺序显示在按钮上，然后随机布局按钮；二是先将图像的顺序随机打乱，然后按打乱后的次序顺序显示在按键上。本案例中使用后一种方法。

具体实现的思路是：将上一步获得的子图像动态数组 subimage 打乱顺序，使用 Collections 类中的乱序方法 shuffle()实现。如下所示。

```
BufferedImage firstImage = subimage.remove(0); //移除第一个图片
Collections.shuffle(subimage);          //对图像动态数组乱序排列
subimage.add(0,firstImage);             //再将移除的第一个子图像添加到数组当中
```

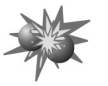
注意，在上面的代码中，首先移除数组中第一个图片，这是因为，第一张图片处在原始图像的边缘，它是用来对应空白按钮的，所以不应该被乱序。移除第一个图片以后，对剩余的图片乱序排列，再把第一张图片加到索引为 0 的位置。那么这时的 subimage 就是乱序以后的图像序列了。

25.3.3 实现图片的移动

在游戏操作中，通过单击与空白网格相邻的网格，就可以将该网格中显示的图片移动，那么就可以用一个按钮代表一个网格，然后让每个按钮显示一个小图片，最后将这些按钮添加到一个面板中，面板采用 GridLayout 网格布局，就可以实现如图 25-1 所示的效果了。关键要解决的是，如何实现单击图片按钮以后图片的移动。这需要弄清两个问题：

- 单击图片按钮时，首先要进行判断，此按钮是否与空白按钮相邻。只有与空白按钮相邻的图片按钮才能移动。
- 如何实现图片移动。

确定如何判断被单击的网格与空白的网格是否相邻，即被单击的按钮和显示空白图片的按钮是否相邻。如果两个按钮相邻，一种是它们在同一行，并且相差 1 列。另一种是它们在同一列，并且相差一行。如果两个按钮相邻并且在同一行，则这两个按钮的行索引相



减得 0，列索引相减的绝对值为 1；同理，如果两个按钮相邻并且在同一列，则这两个按钮的列索引相减得 0，行索引相减的绝对值为 1。

至此，可以得出一个规律，即如果两个按钮相邻，那么这两个按钮的行索引相减的绝对值与列索引相减的绝对值的和恒等于 1。可以通过 java.lang.Math 类的 abs()方法获得一个数的绝对值。其代码如下所示。

```
//判断被单击按钮与空白按钮是否相临
if(Math.abs(clickRow-emptyRow)+Math.abs(clickCol-emptycol) == 1){
    ……
}
```

对于第二个问题，如何实现图片的移动，有两种方案：一是将图片和按钮绑定，然后改变按钮在面板中的位置；另一种是固定按钮在面板中的位置，然后改变按钮上显示的图片。本案例采用第二种方案，即按钮位置固定，当单击与空白图片按钮相邻的按钮时，则令显示空白图片的按钮显示被单击按钮显示的图片，然后令被单击的按钮显示空白的图片。

既然每个按钮的位置是固定的，则可以将每个按钮所在行和列的信息保存到每个按钮对象中。在本例中，通过继承自 JButton 的一个自定义按钮类 ImageButton，为其增加两个代表自身所在行和列的整数属性 row 和 col，并提供读取的 get 和 set 方法。

25.3.4　参考图像的缩放

在图 25-1 中，右边的参考原始图像，是在内存中经过对原始图像的缩放以后，由 JLabel 标签对象显示的。要实现图像的缩放功能，利用 java.awt.geom 包中的 AffineTransform 类。其实现代码如下。

```
AffineTransform transform = new AffineTransform(); //声明一个仿射变换过滤器
```

然后调用该类中的 setToScale()方法来设置图像在 *X*、*Y* 方向上的缩放比例。其实现代码如下。

```
double scaleX=0.5;                          //声明在 X 轴上的缩放比例
double scaleY=0.5;                          //声明在 Y 轴上的缩放比例
transform.setToScale(scaleX,scaleY);        //设置在 X 轴和 Y 轴上的缩放比例
```

接着，使用仿射变换过滤器就可以变换源图像了。该类是 java.awt.image 包中的 AffineTransformOp 类。创建此过滤器的代码如下。

```
AffineTransformOp ato = new AffineTransformOp(transform,null);
```

第一个参数为变换过滤器，第二个参数通常为 null。最后，创建一个目标缓冲图像并清空，然后使用过滤器的 filter()方法，将原始图像按设置的过滤类型过滤到目标图像中。代码如下所示。

```
//另创建一个空的缓冲区图像，大小和类型与 srcImage 相同
image = new BufferedImage(srcImage.getWidth(this)/2,       //目标图像的宽
                          srcImage.getHeight(this)/2,   //目标图像的高
                          BufferedImage.TYPE_INT_ARGB); //目标图像的类型
Graphics2D g2d = image.createGraphics();   //获得目标图像的绘图环境
g2d.clearRect(0,0,image.getWidth(this),image.getHeight(this));
                                           //清空 image 缓冲区图像中的数据
//利用过滤器过滤源图像，并将得到解决的处理后图像数据存入目标图像 image 中
ato.filter(srcImage,image);
```

其中 image 是一个 BufferedImage 对象，它保存的就是缩放以后的图像。

25.4 开发步骤

解决了游戏程序的主要算法问题，下面就来实现拼图游戏。具体的实现步骤如下。

1. 自定义按钮类设计：创建能保存自身位置信息的自定义的按钮类 ImageButton。
2. 游戏主窗体设计：构造窗体类 PuzzleGame。
3. 菜单栏设计：构造菜单栏，并实现事件响应功能。
4. 监听器设计：创建游戏事件监听器，实现游戏功能。
5. 显示参考图形的设计：创建自定义面板组件类 ModelPane。

25.4.1 自定义按钮类设计：ImageButton 类

首先创建一个 ImageButton 类，该类继承自 JButton 类，为该类增加两个属性 row 和 col，都是整型变量，用来保存该按钮对象所在的行和列。同时，为这两个属性增加读取和设置的 get 和 set 方法。具体代码如下。

```
import javax.swing.JButton;
import javax.swing.ImageIcon;
public class ImageButton extends JButton{
     private int row = 0;           //声明用来保存所在行的变量
     private int col = 0;           //声明用来保存所在列的变量
     public ImageButton(){}         //默认构造器
     public ImageButton(ImageIcon image){
                                    //带有参数的构造器，参数为按钮上要显示的图片
          super(image);             //调用超类的构造器
     }
     public void setRow(int row){   //设置按钮所在行的值
          this.row = row;
     }
     public void setCol(int col){   //设置按钮所在列的值
          this.col = col;
     }
     public int getRow(){           //获取按钮所在的行
          return this.row;
     }
     public int getCol(){           //获取按钮所在的列
          return this.col;
     }
}
```

对于属性进行读写的 get 和 set 方法，又称为“读写器”。

25.4.2 游戏界面布局：构造窗体类 PuzzleGame

接着创建一个窗体类，该类继承自 JFrame 类，为程序的主窗体类。然后在该类中声明拼图按钮所在面板和显示参考图片的面板对象、菜单栏对象等。编写该类的构造器，在构造器中调用相应的方法，完成图像的加载和拆分、创建显示图像的按钮，并设置窗体在屏幕中显示的位置。其代码如下所示。

```
import javax.swing.*;
import java.awt.*;
```



```
import java.awt.event.*;
import java.io.*;
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import javax.swing.border.TitledBorder;
import java.util.Collections;
import java.util.ArrayList;
public class PuzzleGame extends JFrame{
     public final static int TYPE =            //定义游戏难度类型，可扩展
     public static final int [] level = {50, 100};
                          //设置拆分图像的大小(像素)。大小决定了游戏的难易度
     int len;             //拆分以后小图像的长度和宽度（以像素为单位）
     int row,col;                              //拼图的行数和列数
     private JPanel centerPane;                //声明拼图按钮所在面板
     private JPanel modelPane;                 //声明显示参考图像的面板
     private BufferedImage image = null;       //要拼接的原图
     private JMenuBar mBar;                    //声明菜单栏对象
     private ImageButton emptyButton;          //声明空白按钮对象
     private String filename = "background/iq.jpg";    //要拼接的图片的路径
     public PuzzleGame(){
          super();                             //调用超类 JFrame 的构造器
          setResizable(false);                 //将窗体设置为不可改变大小
          setTitle("拼图游戏");                 //设置窗体的标题
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                  //设置关闭窗体时退出程序
          this.len = level[TYPE];                 //设置要拆分的图像的大小
          this.image = loadImage();               //加载原始图像
          setRowAndCol();                         //设置要拼接的行和列
          this.centerPane = this.createCenterPane();   //创建拼接图面板
          this.getContentPane().add(centerPane,BorderLayout.CENTER);
          this.modelPane = new ModelPane(this.image);//创建显示参考图像的面板
          this.getContentPane().add(modelPane,BorderLayout.LINE_END);
          this.mBar = this.createMenuBar(); //创建菜单栏
          this.setJMenuBar(mBar);           //设置菜单栏
          this.pack();
          this.setLocation(                 //设置程序窗体出现在屏幕中央位置
                this.getToolkit().getScreenSize().width/2 - this .getWidth()/2,
                this.getToolkit().getScreenSize().height/2 - this.getHeight()/2
          );
          this.setVisible(true);
     }
}
```

一般用户的习惯，程序启动后，窗体界面应位于屏幕的中央。在上面的代码中，通过调用窗体对象的 setLocation()方法设置窗体出现的位置，通过 ToolKit 对象的 getScreenSize()方法获得屏幕的尺寸大小，然后再通过其属性 width 和 height 分别获得屏幕的宽和高，再根据窗体本身的大小，计算出其应出现在屏幕中的左上角的坐标位置。

25.4.3　加载和拆分图像

在 PuzzleGame 类中编写一个实现图像加载的方法 loadImage()，其返回类型为 BufferedImage 对象。在构造器中调用此方法加载图像并赋给实例变量 image。该方法定义如下。

```
//加载图像
public BufferedImage loadImage(){
     BufferedImage image = null;
     try{
          image = ImageIO.read(new File(filename));
     }catch (Exception e){
          System.out.println("加载图像错误！");
     }
     return image;
}
//设置拼图的行数和列数
public void setRowAndCol(){
     //如果加载图像成功，即不为 null
     if(this.image != null){
           this.col = image.getWidth() / this.len;
           this.row = image.getHeight() / this.len;
     }else{
           this.col = 0;
           this.row = 0;
     }
}
```

上面的代码中的 setRowAndCol()方法，用来在加载图像以后，根据加载图像的大小和拆分图像的大小，计算需要拆分成多少行和多少列的子图像。加载图像以后，使用如下的方法拆分图像：

```
//拆分图像
public ArrayList<BufferedImage> dividImage(BufferedImage image)
{
     ArrayList<BufferedImage> subimage = new ArrayList<BufferedImage>
(this.row*this.col);
     for (int i = 0; i <this.row; i++)
           for (int j = 0; j <this.col; j++)
                subimage.add((i*this.col + j),image.getSubimage(j*len,
i*len, len, len));
     BufferedImage firstImage = subimage.remove(0); //移除第一个图片
     Collections.shuffle(subimage);              //对图像动态数组乱序排列
     subimage.add(0,firstImage);
     return subimage;
}
```

方法 dividImage()用来对传进来的参数 image 进行拆分，并将拆分后的图像保存在一个动态数组中，使用 Collections 类的乱序方法 shuffle()对此动态数组乱序排列，然后返回此动态数组。在下一步创建显示图片的按钮的过程中，就可以提取此数组中的图片显示在按钮上了。

25.4.4 创建拼接图面板

接着，编写方法 createCenterPane()，在这个方法中，创建一个 JPanel 对象作为拼接图面板，并将其设置为 GridLayout 布局。其代码如下所示。

```
//创建拼接图面板
private JPanel createCenterPane(){
     JPanel centerPane = new JPanel();
     centerPane.setBorder(new TitledBorder(null,"",
```



```
                    TitledBorder.DEFAULT_JUSTIFICATION,
                    TitledBorder.DEFAULT_POSITION,null,null)); //为面板添加边框
            centerPane.setLayout(new GridLayout(row,col,0,0));
                                                  //设置拼图面板的大小（行和列数）
            JButton[][] imageButton = this.createImgButton(); //创建要拼接的按钮
            this.addButton(imageButton,centerPane);          //向面板中添加按钮
            return centerPane;
        }
```

在 createCenterPane()方法中调用的 createImgButton()方法，用来创建布局在面板中的按钮组件，这是一个二维数组，元素为自定义的 ImageButton 按钮。在创建每一个按钮的过程中，保存其所在的行和列，并为每一个按钮添加事件监听器。然后此方法返回创建的数组对象。其实现代码如下所示。

```
        //创建按钮对象数组
        public ImageButton[][] createImgButton(){
            ArrayList<BufferedImage> images = this.dividImage(this.image);
                                          //拆分原始图像，并保存到数组当中
            ImageButton[][] imageButton = new ImageButton[this.row][this.col];
            for(int row =0 ;row<this.row; row++){
                for(int col =0;col<this.col; col++){
                    imageButton[row][col] = new ImageButton(new ImageIcon
(images.get(row*this.col+col)));
                    imageButton[row][col].setRow(row);
                    imageButton[row][col].setCol(col);
                    imageButton[row][col].setPreferredSize(new Dimension
(len,len));                                   //设置按钮的首选大小
                    imageButton[row][col].addActionListener(new ImgButtom
Action());
                }//end for
            }//end for
            imageButton[0][0].setIcon(null);   //将第一个按钮设为不带图像的按钮
            emptyButton=imageButton[0][0];     //将空白按钮对象指向不带图像的按钮
            return imageButton;
        }
```

在创建按钮数组对象以后，再编写并调用方法 addButton()，在该方法中，遍历 createImgButton()方法返回的按钮数组，并将其一一添加到拼接面板当中。其实现代码如下所示。

```
    //向组件中添加按钮的方法
        public void addButton(JButton[][] imageButton,JPanel centerPane){
            for(int i=0;i<row;i++)
                for(int j=0;j<col;j++)
                    centerPane.add(imageButton[i][j]);
        }
```

25.4.5　创建显示参考图像的面板

为了使玩家在玩游戏时随时有参考图像可供对比，需要在拼接图面板的右边添加显示参考图像的面板。在本案例中，专门自定义了一个类 ModelPane，它继承自 JPanel 类。在构造 ModelPane 时，传递进要显示的参考图像参数。其实现代码如下所示。

```
    import java.awt.*;
    import javax.swing.*;
    import java.awt.geom.AffineTransform;
```

```
import java.awt.image.AffineTransformOp;
import java.awt.image.BufferedImage;
import javax.swing.border.TitledBorder;
public class ModelPane extends JPanel{
     AffineTransform transform = new AffineTransform(); //声明一个仿射变换过滤器
     private double scaleX=0.5;                  //声明在 X 轴上的缩放比例
     private double scaleY=0.5;                  //声明在 Y 轴上的缩放比例
     private BufferedImage destImage;            //声明源图像和缩放以后的图像
     public ModelPane(BufferedImage srcImage){
          super();
          destImage = filterSrcImage(srcImage);
                                   //对原始图像进行缩放，并返回缩放后的图像
          JLabel modelLabel = new JLabel(new ImageIcon(destImage));
          JLabel infoLabel = new JLabel("原始图像",SwingConstants.CENTER);
          this.setLayout(new BorderLayout());
          this.setBorder(new TitledBorder(null,"",
                    TitledBorder.DEFAULT_JUSTIFICATION,
                    TitledBorder.DEFAULT_POSITION,null,null));
                                   //为面板添加边框
          this.add(infoLabel,BorderLayout.PAGE_START);
          this.add(modelLabel,BorderLayout.CENTER);
     }
}
```

在构造器中，调用 filterSrcImage()方法创建一个经过缩放以后的图像。该方法在构造器之后定义，为类 ModelPane 的实例方法，其实现代码如下所示。

```
//对指定图像进行缩放
public BufferedImage filterSrcImage(BufferedImage srcImage){
     BufferedImage image;
     AffineTransform transform = new AffineTransform();
                                             //声明一个仿射变换过滤器
     double scaleX=0.5;                      //声明在 X 轴上的缩放比例
     double scaleY=0.5;                      //声明在 Y 轴上的缩放比例
     transform.setToScale(scaleX,scaleY);    //设置在 X 轴和 Y 轴上的缩放比例

     //另创建一个空的缓冲区图像，大小和类型与 srcImage 相同
     image = new BufferedImage(srcImage.getWidth(this)/2,
                               srcImage.getHeight(this)/2,
                               BufferedImage.TYPE_INT_ARGB);
     Graphics2D g2d = image.createGraphics();
                                          //获取 image 对象的绘图上下文环境
     //清空 image 缓冲区图像中的数据
     g2d.clearRect(0,0,image.getWidth(this),image.getHeight(this));
     //创建一个仿射变换过滤器
     AffineTransformOp ato = new AffineTransformOp(transform,null);
     //利用过滤器过滤源图像，并将得到解决的处理后图像数据存入 image 中
     ato.filter(srcImage,image);
     return image;
}
```

filterSrcImage()方法接收一个原始的 BufferedImage 对象，并对其进行变换，将变换后的图像赋给一个新的 BufferedImage 对象 image，然后返回 image。在构造器中调用 filterSrcImage()方法以后，将获得的 image 对象赋给 BufferedImage 类型变量 destImage。然后在构造器中构造一个显示 destImage 图片的标签 modelLabel，并显示在 ModelPane 面板



的中间位置。

25.4.6　创建菜单栏

在上面两步中，创建拼接图面板和显示参考图像的面板，并添加到窗体的内容面板。然后，就要调用 createMenuBar()方法创建菜单栏，并调用 setJMenuBar()为窗体设置菜单栏。创建菜单栏的方法定义如下。

```
//创建菜单栏
private JMenuBar createMenuBar(){
     JMenu[] m = {new JMenu("开始(B)"),new JMenu("关于(A)")};
                                          //创建两个菜单
     JMenuBar mBar = new JMenuBar();      //创建一个菜单栏对象
     //this.setJMenuBar(mBar);
     mBar.add(m[0]);                      //为菜单栏添加第一个菜单
     mBar.add(m[1]);                      //为菜单栏添加第二个菜单
     m[0].setMnemonic('B');               //设置快捷键
     m[1].setMnemonic('A');               //设置快捷键
     initMenuBegin(m);                    //初始化“开始”菜单
     initMenuAbout(m);                    //初始化“关于”菜单
     return mBar;                         //返回创建的菜单栏
}
```

其中将初始化子菜单的方法封装到两个单独的方法 initMenuBegin()和 initMenuAbout()中，这样使得程序结构更加清晰。initMenuBegin()方法用来初始化“开始”菜单，其方法的定义如下。

```
private void initMenuBegin(JMenu[] m){
          JMenuItem [] mI = {              //创建子菜单项
              new JMenuItem("新开局(N)"),
              new JMenuItem("退出(E)")
        };
          mI[0].setMnemonic('N');          //设置子菜单项的快捷键
          mI[1].setMnemonic('E');          //设置子菜单项的快捷键
          mI[0].setAccelerator(KeyStroke.getKeyStroke("ctrl N"));
                                           //设置子菜单项的加速键
          mI[1].setAccelerator(KeyStroke.getKeyStroke("ctrl W"));
                                           //设置子菜单项的加速键
          mI[0].addActionListener(new ActionListener(){
                                           //为“新开局”菜单项添加事件监听器
                public void actionPerformed(ActionEvent e){
                     menuNewClick();   //调用方法进行响应
               }
            }
          );
          mI[1].addActionListener(new ActionListener(){
                                           //为“退出”菜单项添加事件监听器
               public void actionPerformed(ActionEvent e){
                     System.exit(0);   //退出程序
               }
               }
          );
          m[0].add(mI[0]);                 //将第一个子菜单项添加到菜单中
```

```
            m[0].add(mI[1]);                    //将第二个子菜单项添加到菜单中
            m[0].insertSeparator(1);            //插入分隔线
        }
        //响应用户单击“新开局”菜单项的方法
        private void menuNewClick(){
              this.getContentPane().remove(centerPane);  //移除当前拼接图面板
              this.centerPane = this.createCenterPane();
                                                         //创建一个新的拼接图面板
              this.getContentPane().add(this.centerPane,BorderLayout.CENTER);
             this.validate();                            //重新布置组件
    }
```

可以看到，上面为菜单项添加的事件监听器，使用了内部匿名类。在改变了容器中的组件以后，一定要调用 validate()方法，重新布置并显示组件。initMenuAbout()方法用来初始化“关于”菜单，其定义如下。

```
    private void initMenuAbout(JMenu[] m){
        JMenuItem [] mI = {new JMenuItem("游戏说明(H)"),new JMenuItem("关
于作者(A)")};
            mI[0].setMnemonic('H');                  //设置子菜单项的快捷键
            mI[1].setMnemonic('A');                  //设置子菜单项的快捷键
            mI[0].setAccelerator(KeyStroke.getKeyStroke("F1"));
            mI[0].addActionListener(new ActionListener(){
                                                     //当单击“游戏说明”菜单时
                public void actionPerformed(ActionEvent e){
                    String help0 = "移动小图片，组合成如右边所示原始图像.\n\n";
                    String help1 = "操作方法 :在邻近空白图片的图片上单击,将其位置
与空白图片位置互换.";
                    JOptionPane.showMessageDialog(null,  help0  +  help1);
    //显示消息对话框
                }
            });
            mI[1].addActionListener(new ActionListener(){
                                                     //当单击“关于作者”菜单时
                public void actionPerformed(ActionEvent e){
                    String version = "版本: 1.0\n";
                    String author = "作者: 辛立伟  张帆\n";
                    String email = "E-mail: hn-zjxlw2002@163.com";
                    JOptionPane.showMessageDialog(null, version + author
+ email);
                }
            }
            );

            m[1].add(mI[0]);                         //将第一个子菜单项添加到菜单中
            m[1].add(mI[1]);                         //将第二个子菜单项添加到菜单中
    }
```

在上面的这个方法中，同样使用匿名内部类作为菜单项的事件监听器，这是常用的一种方法。

25.4.7 实现按钮事件监听器类

在创建显示图片的按钮数组时，为每一个按钮都添加了事件监听器。这里，需要创建



按钮的事件监听器类 ImgButtomAction，这是 PuzzleGame 类的内部类，这样做的好处是可以方便地在事件响应方法中布局窗体中的按钮。其实现代码如下所示。

```
        //内部类，充当监听器
        class ImgButtomAction implements ActionListener{
             public void actionPerformed(ActionEvent e){
                  ImageButton clickButton = (ImageButton)e.getSource();
                                                          //获得被单击的按钮对象
                  int clickRow = clickButton.getRow();  //获得被单击按钮所在的行
                  int clickCol = clickButton.getCol();  //获得被单击按钮所在的列
                  int emptyRow = emptyButton.getRow();  //获得空白按钮所在的行
                  int emptycol = emptyButton.getCol();  //获得空白按钮所在的列
                  //判断被单击按钮与空白按钮是否相临
                  if(Math.abs(clickRow-emptyRow)+Math.abs(clickCol-emptycol)
== 1){
                       //将被单击按钮的图片移动到空白按钮上
                       emptyButton.setIcon(clickButton.getIcon());
                       //将被单击的按钮的图片设置为空
                       clickButton.setIcon(null);
                       //将空白按钮指向被单击的按钮，设置新的空白按钮
                       emptyButton = clickButton;
                  }
             }
        }
```

关于图片互换的算法，请参见“算法设计”一节。

25.4.8　设置窗体在屏幕中出现的位置

在 PuzzleGame 类布局的最后，在构造器中需要设置窗体出现的位置，使其运行起来更符合用户的习惯。这里，将窗体设置为出现在屏幕的中央。其实现代码如下所示。

```
        this.pack();                              //设置组件以首选大小排列
        this.setLocation(                         //设置程序窗体出现在屏幕中央位置
             this.getToolkit().getScreenSize().width/2 - this.getWidth()/2,
             this.getToolkit().getScreenSize().height/2 - this.getHeight()/2
        );
        this.setVisible(true);                                  //显示窗体
```

在这里，调用 Frame 类的 getToolkit()方法，获得一个 ToolKit 对象，它代表窗体所在的系统环境。ToolKit 类有个静态方法 getScreenSize()，可以返回屏幕的大小。在这里，将窗体的放置在屏幕的中央位置。最后，调用 setVisible(true)方法，显示窗体。这个方法一定要调用，否则窗体默认是不显示的。

25.4.9　编写 main 方法

布局和事件响应代码都已经实现了，最后一步，就是编写 main()方法，创建 PuzzleGame 类的实例了。main()方法非常简单，代码如下所示。

```
    public static void main(String[] args){
         PuzzleGame game = new PuzzleGame();            //创建本类的对象
    }
```

25.4.10 运行测试

至此，一个简单的拼图游戏就编写完成了。打开控制台，使用下面的命令编译并运行，试着玩一玩自己编写的游戏，体会一下小小的成就感吧！

```
javac PuzzleGame.java
java PuzzleGame
```

在编译和运行之前，确认在与 PuzzleGame.java 源文件相同的目录下，存在一个 background 文件夹。如果不存在的话，创建它。然后将一个“300 像素*400 像素”大小的 jpg 图片放在此文件夹下，并命名为“iq.jpg”。启动程序，可以看到如图 25.1 所示程序界面。

25.5 打包部署

虽然可以成功地运行 PuzzleGame 游戏程序，但是美中不足的是，还必须在命令行下输入复杂的命令才能运行。不过，Java 提供了名为 jar 的工作软件，可以将用 Java 语言编写的桌面应用程序打包为双击就可以执行的程序。其打包步骤如下所示。

① 使用“javac PuzzleGame.java”命令，获得所有的.class 文件。

② 在与.class 类文件同一目录下，使用记事本创建一纯文本文档文件，命名为 MANIFEST.MF。编辑 MANIFEST.MF 文件如下。

```
Manifest-Version: 1.0
Created-By: 1.6.0_04 (Sun Microsystems Inc.)
Main-Class: PuzzleGame
```

创建 MANIFEST.MF 文件一定要注意：每一行的冒号和后面的内容之间要有空格；每一行的结尾一定要用回车键换行。其中 Main-Class：后面一定要跟带 main()方法的类的类名。

③ 在同一目录下，使用如下的命令打包文件：

```
jar cvfm PuzzleGame.jar MANIFEST.MF .
```

注意，最后的圆点代表将当前目录的所有文件打包成 PuzzleGame.jar 文件（文件名可任意取）。运行此命令之后，在当前目录下，会发现出现了一个名为 PuzzleGame.jar 的文件。

④ 将上一步创建的档案文件 PuzzleGame.jar 与 background 放在同一目录下，然后双击运行，会发现程序 PuzzleGame 已经启动，出现图 25-1 所示运行界面，而不再需要使用命令行启动了。

恭喜，现在可以将 PuzzleGame.jar 与 background 放在同一目录下，发给朋友与你一起分享游戏的快乐吧！不过要提醒的是，这种方式打包的可执行程序，必须部署在安装有 Java 虚拟机的计算机上才能运行。

25.6 小结

在本章中，通过一个完整的拼图游戏的开发过程，介绍了创建 Java 应用程序的步骤和方法。重点对游戏的算法设计和代码实现做了详细的说明。在本例中，综合运用到了 Java 的 Swing 编程、继承类、集合和泛型的使用、事件处理等知识点，让读者真正达到独立开发的能力和综合运用多种知识的能力。其中的算法设计及代码中图像处理部分，是本章学习的难点，读者可参考书中提供的示例源代码，实际上机调试并运行，以帮助理解。

下一章将介绍一个基于 TCP 协议开发的双向通信的网络聊天室。

第26章　案例2：心心网络聊天室

通常来说，网络程序的开发是比较复杂的。但是Java语言提供了强大的网络编程API，程序员可以使用这些网络API很容易地开发出复杂的网络应用程序来。本章通过一个网络聊天程序的开发案例，详细讲解综合运用Java基于TCP的C/S结构，以实现网络上的两个程序通过一个双向的通信连接实现数据交换的功能，帮助读者联系实际掌握网络编程的内容。



26.1　软件简介

聊天室是人们现在在Internet上交流的重要场所，随着Internet技术的不断发展，出现了很多不同种类的聊天室，用于方便人们的交流。甚至现在有了用于聊天的软件和专门的聊天服务器，可见聊天程序的作用之大。

26.1.1　应用背景

既然聊天室这么实用，使用的人群这么广，那么编写一个实用的聊天室，给用户提供一个基于网络的交流平台就特别有意义了。本章就通过使用Java语言，重点是其Socket套接字编程和多线程技术，编写一个实用的聊天室。通过网络聊天室程序的开发，进而从中提高读者综合运用理论知识（主要是Socket网络编程技术）的能力，同时读者的实际动手能力也能得到很大提高。

26.1.2　需求分析

本案例应用中，是在服务器上运行一个永久的程序，它可以接收来自其他多个客户端的请求，提供相应的服务。为了实现服务器方给多个客户提供服务的功能，利用多线程来实现多客户机制。服务器总是在指定的端口上监听是否有客户请求，一旦监听到客户请求，服务器就会启动一个专门的服务线程来响应该客户的请求，而服务器本身在启动完线程之后马上又进入监听状态，等待下一个客户的到来。

心心聊天室主要由两部分组成：服务器端（Server）和客户端（Client）。在客户端的图形界面中，可以显示所有出现在聊天室中的用户名称，在“聊天记录”选项区域中显示聊天的内容以及用户进入和离开聊天室的信息。

因为硬件条件的限制，本案例的两个客户端程序也是在服务器所在的计算机上运行的，所以其“服务器 IP”文本框中采用了默认值 LocalHost，表示本机的 IP 地址。

至于客户端与服务器之间的通信，主要是通过 Socket 套接字来创建服务器和客户端的连接，一般 Socket 类来初始化服务器名和端口号。有了连接后，就必须使用 Java 提供的两个对象（PrintStream，BufferedReader）来读入和写出 Socket，这是使用 Socket 类提供的两个方法（getInputStream()，getOutputStream()）来使得以上的两个对象分别从 Socket 中读数据和往 Socket 中写数据，使得用户发送的数据可以通过 Socket 来传输。

下面分析心心聊天室的基本功能。

1．用户连接

用户填写一个用户名，单击【连接】按钮，便可以连接服务器端。图 26.1 为客户端用户在用户名对应的文本区域中输入用户名并单击【连接】按钮的效果图。

2．聊天面板

新用户“笑笑”和“哈哈”分别在用户名文本区输入自己的名字并单击【连接】按钮进入聊天室，首先在【在线用户】区域显示所有进入聊天室的用户名称，当用户在聊天界面最下面的聊天区域输入聊天信息并按下 Enter 键，则所有用户的聊天内容都会显示在【聊天记录】区域。下面两个图 26.2 与 26.3 为两个用户“笑笑”和“哈哈”进入聊天室并进行聊天的界面。

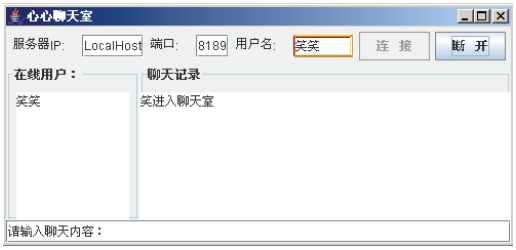


图 26.1　用户连接

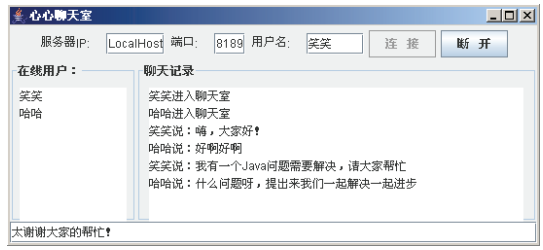


图 26.2　用户笑笑进入聊天室并发送聊天信息

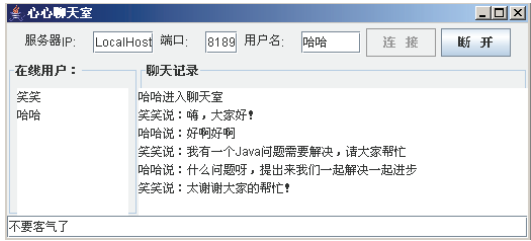


图 26.3　用户哈哈进入聊天室并发送聊天信息

表 26.1 显示了心心聊天室程序中使用的一些主要的类和方法。

表 26.1　心心聊天室用到的类和方法

名　　称	使 用 类	使用方法
心心聊天室所用的类和方法	ChatServer 类	ChatServer()：创建服务器
		run()：运行服务器端，接收客户端的连接请求
		removeConnectionList()：删除连接
		returnUsernameList()：返回用户列表
		broadcastMessage()：发送消息



续表

名　　称	使 用 类	使用方法
心心聊天室所用的类和方法	ChatHandler 类	ChatHandler()：创建线程用来响应客户端
		sendMessage()：将服务器发送的信息传递给客户端
		setupUserName()：服务器告知其他客户端新的客户连接
		extractMessage()：处理客户端的信息，并把该信息发送给其他客户端
		disconnectClient()：断开服务器连接，并返回用户名
		run()：获取客户输入的信息
	ChatClient 类	createComponents()：创建图形用户界面
		actionPerformed()：用于响应 connect、disconnect 和 messageField 的动作
		disconnectFromServer()：与服务器断开
		connectToServer()：连接服务器
		extractMessage()：处理服务器发来的信息，并做出适当的响应
		updateList()：更新在线用户名
		run()：读取服务器发来的信息
	Message 类	getType()：返回信息的类型
		getBody()：返回信息的具体内容
		isValid()：返回信息是否有效



26.2　设计思路与原理

心心网络聊天室是一个 C/S 结构的聊天程序，是使用 Socket 实现服务器和客户端的连接，它的一般连接过程是这样的：

- 服务器程序监听某一端口，等待客户接入；同时构建一个线程类，准备接管对该客户的服务。
- 当一个客户端连接了服务器后，创建一个线程类对象，专门处理这个客户端与服务器之间的信息传递，然后主程序继续监听端口。
- 当一个客户端向服务器发送信息后，服务器将这个信息传递给所有连接的客户端，从而实现信息共享。

其具体实现过程需要完成三部分工作：服务器实现、客户端实现、用户界面设计实现。

26.2.1　服务端

在服务器端，用 Java 语言中的服务器类——ServerSocket，使用端口号作为参数来创建服务器对象。例如：

```
final static int thePort=8189;
ServerSocket theServer;
theServer=new ServerSocket(thePort);
```

这几条语句创建了一个服务器对象，这个服务器使用 8189 号端口。当一个客户端程序建立一个 Socket 连接，所连接的端口号为 8189 时，服务器对象 Server 便响应这个连接。另外对于这几条语句也可以用以下的一条语句实现。例如：

```
ServerSocke theServer =new ServerSocket（8189）;
```

接下来用 theServer.accept()方法创建一个 Socke 对象，服务器端便可以利用这个 Socket

对象与客户端进行通信。例如：

```
Socket theSocket=theServer.accept();
```

紧接着用 ChatHandler 类中的构造方法 ChatHandler(Socket socket,ChatServer parent)创建一个对象 chatHandler：

```
ChatHandler chatHandler=new ChatHandler(theSocket,this);
```

下面是方法 ChatHandler (Socket socket,ChatServer parent)中的语句，用于获得输入流和输出流，并进行封装。例如：

```
    BufferedReader in=new BufferedReader(new InputStreamReader(theSocket.
getInputStream()));
    PrintWriter out=newPrintWriter(newOutputStreamWriter(theSocket.
getOutputStream()),true);
```

随后，就可以使用 in.readLine()方法得到客户端的输入，也可以使用 out.println()方法向客户端发送数据。从而可以根据程序的需要对客户端的不同请求进行回应。

26.2.2 客户端

客户端比较简单，核心代码在 ChatClient 类中。在该类中有一个 Socket 类的对象 theSocket，与服务端进行连接。连接好建立并得到这个对象后，就可以用前面介绍的方法实现数据的输入和数除。例如：

```
Socket theSocket;
BufferedReader in;
PrintWriter out;
theSocket=new Socket(ip,port);
in=new BufferedReader(new InputStreamReader(theSocket.getInputStream()));
out=new PrintWriter(new OutputStreamWriter(theSocket.getOutputStream()),true);
```

为了简化设计，用户名在客户端控制，服务器端只进行简单的接收信息和“广播”此信息，不进行名字校验，也就是说，可以有同名客户登录到服务端。这个程序设计虽然简单，但是已经具备了聊天室的最基本的功能。

26.2.3 建立用户界面

在本案例中，建立用户界面是在客户端程序中实现的，大家也可以利用前面学过的内容，根据自己的喜好建立自己的用户界面。

经过以上三个步骤就可以建立一个比较简单的对话程序。但是为了使这个程序能够实现多个用户同时在线聊天，应把服务器端的程序用多线程实现，这样可以使服务器为多个客户端服务。这正是在前面所讲到的 ChatHandler 线程类的作用。例如：

```
try{
    System.out.println("服务器已建立！");
    while(numbers<10){
        Socket theSocket=theServer.accept();
        ChatHandler chatHandler=new ChatHandler(theSocket,this);
        chatters[numbers]=chatHandler;
        numbers++;
     }
}catch(IOException e){}
```

上面的程序片段循环检测是否有客户连接到服务器上，如果有，则创建一个线程来服务这个客户，这个线程的名称是 ChatHandler，它是 Thread 类的子类。

26.3　业务流程图

通过前面对心心网络聊天室的需求介绍及设计思路的讲解，读者对这个聊天室的业务流程也应该心中有数了吧。心心网络聊天室的业务流程图如图 26.4 所示。

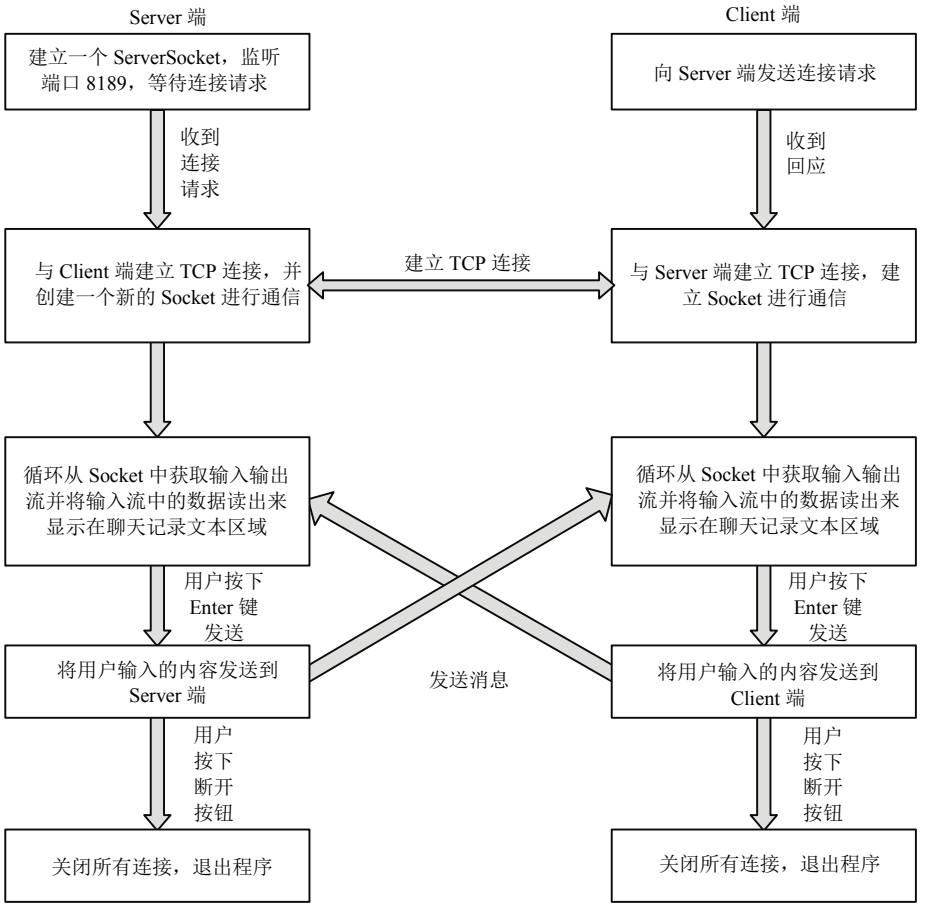


图 26.4　心心网络聊天室的业务流程图

26.4　应用程序实现

了解了总体设计与概要设计，开始进行实际编程，在这里，将编写 4 个类来实现“网络聊天工具”程序。具体的实现步骤如下。

① 创建 ChatServer 类：ChatServer 类用来建立服务器。
② 创建 ChatHandler 类：ChatHandler 类用来处理其对应客户端与服务器之间的通信。
③ 创建 ChatClient 类：ChatClient 类用来建立客户端。
④ 创建 Message 类：创建 Message 类来对信息进行处理。

26.4.1 建立服务器：ChatServer 类

定义一个 ChatServer 类用来建立服务器，其 IP 地址为运行该程序的计算机的地址，端口设定为 8189。设定该服务器只能处理 10 个线程，也就是说只能同时有 10 个人在线聊天。

ChatHandler 类型数组 chatters 用来保存当前运行的线程。当有客户端与服务器连接后，ChatServer 类会创建一个 ChatHandler 类对象与该客户端对应，并将该 ChatHandler 类对象添加到 chatters 数组中。当有客户端与服务器断开时，将与该客户端对应的 ChatHandler 类对象从 chatters 数组除去。

ChatServer 类的 broadcastMessage()方法用来将获取的信息发送给所有连接的客户端。returnUsernameList()方法用来返回所有连接的客户端的用户名。其程序代码如下。

```
import java.io.*;
import java.net.*;
/*建立服务器端*/
public class ChatServer {
    final static int thePort=8189;                    //设定服务器端口为 8189
    ServerSocket theServer;     //声明服务器端的 Socket
    ChatHandler[] chatters;     //用来保存所有正在运行的 ChatHandler 类对象线程
    int numbers=0;              //记录当前正在运行的 ChatHandler 类对象线程的个数
    /*构造器*/
    public ChatServer(){
        try{
            theServer=new ServerSocket(thePort);    //创建服务器端套接字
            chatters=new ChatHandler[10];//服务器端最多可以同时处理 10 个线程
        }catch(IOException io){}
    }
    /*运行服务器端，接收客户端的连接申请*/
    public void run(){
        try{
            System.out.println("服务器已建立！");
            while(numbers<10){  //判断连接服务器的客户端是否不超过 10 个
                // theServer 调用 accept()方法，响应客户请求并
                //会返回一个 Socket 对象用于同客户进行交互
                Socket theSocket=theServer.accept();
                ChatHandler chatHandler=new ChatHandler(theSocket,this);
                chatters[numbers]=chatHandler;      //将线程添加到数组中
                numbers++;                  //增加客户计数
            }
        }catch(IOException e){}
    }
    /*当某个客户端与服务器端断开连接时，将其从 chatters 数组中去除*/
   public synchronized void removeConnectionList(ChatHandler c){
       int index=0;                         //保存断开连接的服务器所在的下标值
       for(int i=0;i<=numbers-1;i++)
            if(chatters[i]==c)  index=i;    //当找到时，保存其下标值
        for(int i=index;i<=numbers-1;i++){
            chatters[i]=chatters[i+1];      //其后的数组元素一次向前移动一位
            chatters[numbers-1]=null;       //最后一位为 null
            numbers--;                      //当前运行线程减 1
        }
```



```
    }
    /*返回当前在线的用户名*/
    public synchronized String returnUsernameList(){
        String line="";
        for(int i=0;i<=numbers-1;i++)
            line=line+chatters[i].user+":";
                        //将所有用户名保存到 line 中，之间用：隔开
        return line;
    }
    /*将信息发送给每一个客户端*/
    public void broadcastMessage(String line){
        System.out.println("发布信息："+line);
        for(int i=0;i<=numbers;i++)
            chatters[i].sendMessage(line);
    }
    public static void main(String args[]){
        ChatServer app=new ChatServer();    //创建程序对象 app
        app.run();      // app 调用 run()方法用于运行服务器并接收客户端的连接请求
    }
}
```



在选择端口时，必须小心。每一个端口提供一种特定的服务，只有给出正确的端口，才能获得相应的服务。0～1023 的端口号为系统所保留，例如 HTTP 服务的端口号为 80，Telnet 服务的端口号为 21，FTP 服务的端口号为 23，所以在选择端口号时，最好选择一个大于 1023 的数以防止发生冲突。

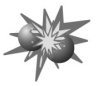
另外，在创建 socket 时如果发生错误，将产生 IOException，在程序中必须对之做出处理。所以在创建 Socket 或 ServerSocket 是必须捕获或抛出例外。

26.4.2　通信处理：ChatHandler 类

ChatHandler 类用来处理其对应客户端与服务器之间的通信，也是一个线程。ChatHandler 类必须继承 Thread 类。ChatHandler 类具体功能是将服务器发送的信息传递到其对应的客户端，同时也把客户端发送的信息传递给服务器。其程序代码如下。

```
import java.net.*;
import java.io.*;
/*对于每个与服务器连接的客户端，创建一个线程来进行处理*/
class ChatHandler extends Thread{
    Socket theSocket;           //保存与本线程相关的 Socket 对象
    BufferedReader in;
    PrintWriter out;
    int thePort;
    ChatServer parent;
    String user = "";           //保存用户名
    boolean disconnect=false;   //保存连接状态
    /*创建线程用来响应该客户端*/
    public ChatHandler(Socket socket,ChatServer parent){
        try{
            theSocket=socket;   //初始化 theSocket 变量
           this.parent=parent; //其对应的服务器为创建的 ChatServer 类的对象
           in=new BufferedReader(new InputStreamReader(theSocket.
```

```
getInputStream()));
                //建立 BufferedReader 对象 in 用于存放从 socket 中取得的输入流
               out=new PrintWriter(new OutputStreamWriter(theSocket.
getOutputStream()),true);
                //从 theSocket 中取得输出流，并构造 PrintWriter 对象 out
               thePort=theSocket.getPort();
               start();                //开始运行线程
            }catch(IOException io){}
        }
        /*将服务器发送的信息传递到该客户端*/
        public void sendMessage(String line){
            out.println(line);
        }
        /*服务器发送信息，告知其他客户端新的客户连接*/
        public void setupUserName(String setname){
            user = setname;
            parent.broadcastMessage("JOIN:"+user);
        }
        /*处理从客户端发来的信息，并发送对应的信息到客户端*/
        public void extractMessage(String line){
            Message messageline;                                    //用来信息处理
            messageline = new Message(line);
            if(messageline.isValid()){
                if(messageline.getType().equals("USER")){  //如果信息为申请连接
                    setupUserName(messageline.getType());
                                                        //运行 setupUsername()方法
                    //向其他所有客户端发送新的用户名列表
                    parent.broadcastMessage("List:"+parent.returnUsernameList());
                }else if(messageline.getType().equals("MESSAGE")){
                                                        //如果信息为聊天信息
                    //向所有客户端发送聊天信息
                    parent.broadcastMessage("MESSAGE:"+user+"说："+messageline.
getBody());
                }else if(messageline.getType().equals("LEAVE")){
                                                        //如果客户端断开连接
                    //调用 disconnectClient()方法，并返回用户名
                    String c=disconnectClient();
                    parent.broadcastMessage("REMOVE:"+c);
                                                        //告知其他客户端该用户离开了
                    parent.broadcastMessage("List:"+parent.returnUsernameList());
                }
            }
            else
                sendMessage("命令不存在！");
        }
        /*与服务器断开，并返回用户名*/
        public String disconnectClient(){
            try{
                in.close();                 //关闭 Socket 输入流
                out.close();                //关闭 Socket 输出流
                theSocket.close();          //关闭 Socket
                parent.removeConnectionList(this);
                                            //将线程从服务器的 chatters 数组中除去
                disconnect=true;
```



```
        }catch(Exception ex){}
        return user;
    }
    /*获取客户输入的信息*/
    public void run(){
        String line,name;
        boolean valid=false;
        try{
            while((line=in.readLine())!=null){
                System.out.println("收到："+line);
                extractMessage(line);
            }
        }catch(IOException io){}
    }
}
```

在该类中 disconnectClient()方法主要实现的是关闭 Socket，即断开服务器。因为每一个 Socket 存在时，都将占用一定的资源，在 Socket 对象使用完毕时，要将其关闭。关闭 Socket 可以调用 Socket 的 close()方法。

在关闭 Socket 之前，应将与 Socket 相关的所有输入/输出流全部关闭，以释放所有的资源。而且要注意关闭的顺序，与 Socket 相关的所有的输入/输出流首先关闭，然后再关闭 Socket。尽管 Java 有自动回收机制，网络资源最终是会被释放的。但是为了有效的利用资源，建议读者按照合理的顺序主动释放资源。

26.4.3　建立客户端：ChatClient 类

ChatClient 类用来建立客户端。首先在 createComponents()方法中创建客户端的图形用户界面。当用户单击【连接】按钮时，调用 connectToServer()创建客户端。当用户单击【断开】按钮时，调用 disconnnectFromServer()方法，关闭客户端。其程序代码如下。

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*建立客户端*/
public class ChatClient extends Thread implements ActionListener{
    JTextField messageField,IDField,ipField,portField;
    JTextArea message,users;
    JButton connect,disconnect;
    String user="";
    String userList[]=new String[10];          //保存在线用户名，最多 10 人
    Socket theSocket;
    BufferedReader in;
    PrintWriter out;
    boolean connected=false;                   //连接状态
    Thread thread;
    public static void main(String args[]){
        JFrame frame=new JFrame("聊天室");      //创建聊天室窗口
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  //关闭窗口
        ChatClient cc=new ChatClient();
        JPanel content=cc.createComponents();
        frame.getContentPane().add(content);
```

```
        frame.setSize(550, 310);                    //设置窗口大小
        frame.setVisible(true);                     //设置 frame 这个组建显示为可见
    }
    /*创建图形用户界面*/
    public JPanel createComponents(){
        JPanel pane=new JPanel(new BorderLayout());
        message=new JTextArea(10,35);               //用来显示聊天内容
        message.setEditable(false);                 // 设置为不可以编辑，也就是只读状态
        JPanel paneMsg=new JPanel();                //用来容纳 message 组件
        paneMsg.setBorder(BorderFactory.createTitledBorder("聊天内容"));
        paneMsg.add(message);
        users=new JTextArea(10,10);                 //用来显示在线用户名
        JPanel listPanel=new JPanel();              //用来容纳 users 组件
        listPanel.setBorder(BorderFactory.createTitledBorder("在线用户："));
        listPanel.add(users);
        messageField=new JTextField(50);            //用户在此输入聊天信息
        IDField=new JTextField(5);                  //用来输入用户名
        ipField=new JTextField("LocalHost");//用来输入服务器的 ip 地址
        portField=new JTextField("8189");           //用来输入服务器的端口号
        connect=new JButton("连   接");             //【连接】按钮
        disconnect=new JButton("断   开");          //【断开】按钮
        disconnect.setEnabled(false);               //【断开】按钮的初始化状态为未激活
        JPanel buttonPanel=new JPanel();            //用来容纳按钮及其标签
        buttonPanel.add(new Label("服务器 IP:"));
        buttonPanel.add(ipField);   //向面板 buttonPanel 中添加组件 ipField
        buttonPanel.add(new Label("端口:"));
                                    //向面板 buttonPanel 中添加标签“端口:”
        buttonPanel.add(portField);
        buttonPanel.add(new Label("用户名:"));
        buttonPanel.add(IDField);
        buttonPanel.add(connect);
        buttonPanel.add(disconnect);
        pane.add(messageField,"South");//设置 messageField 组件在面板中的位置
        pane.add(buttonPanel,"North");
        pane.add(paneMsg,"Center");
        pane.add(listPanel,"West");
        connect.addActionListener(this);
        //把 connect 实例加上一个监听器，当鼠标单击了这个 connect 时就会触发这一个监听器
        disconnect.addActionListener(this);
        messageField.addActionListener(this);
        return pane;
    }
    public void actionPerformed(ActionEvent e){//用于接收操作事件的侦听器接口
        if(e.getSource()==connect){
            //调用 getText()方法获得 IDField 文本区的内容并赋值给变量 user
            user=IDField.getText();
            String ip=ipField.getText();
            //调用 getText()方法获得 portField 文本区的内容，然后
            int port=Integer.parseInt(portField.getText());
            //通过包装类 Integer 的 parseInt（）方法把该内容转换为整型数据并赋值给
变量 port
            if(!user.equals("")&&connectToServer(ip,port,user)){
                disconnect.setEnabled(true);        //设置该对象的启用状态
```



```
                connect.setEnabled(false);          //设置该对象的禁用状态
            }
        }
        if(e.getSource()==disconnect)
            //如果调用 getSource()方法，就能使获得的信息与 disconnect 对象相等
            disconnectFromServer();
        if(e.getSource()==messageField)
            if(theSocket!=null){
                //输出 messageField 文本区的内容
                out.println("MESSAGE:"+messageField.getText());
                messageField.setText("");
            }
    }
    /*与服务器断开*/
    public void disconnectFromServer(){
        if(theSocket!=null){
            try{
                connected=false;
                out.println("LEAVE:"+user);
                disconnect.setEnabled(false);   //设置该对象的禁用状态
                connect.setEnabled(true);       //设置该对象的启用状态
                thread=null;
                theSocket.close();              //关闭 theSocket
            }catch(IOException io){}
            theSocket=null;
            message.setText("");
            users.setText("");
        }
    }
    /*与服务器建立连接，如果成功返回 true，否则返回 false*/
    public boolean connectToServer(String ip,int port,String ID){
        if(theSocket!=null)                 //如果已经建立连接则返回 false
            return false;
        try{
            theSocket=new Socket(ip,port);
            in=new BufferedReader(new InputStreamReader(theSocket.
getInputStream()));
            out=new PrintWriter(new OutputStreamWriter(theSocket.
getOutputStream()),true);
            out.println("USER:"+user);      //向服务器发送信息
            message.setText("");
            connected=true;
            thread=new Thread(this);
            thread.start();
        }catch(Exception e){return false;}
        return true;
    }
    /*处理服务器送来的信息，并做出适当的回应*/
    public void extractMessage(String line){
        System.out.println(line);                          //输出收到的信息
        Message messageline;
        messageline=new Message(line);
        if(messageline.isValid()){
            if(messageline.getType().equals("JOIN")){     //如果有新用户加入
                user=messageline.getBody();
```

```
                message.append(user+"进入聊天室\n");        //添加显示内容
            }
            else if(messageline.getType().equals("LIST"))//更新在线用户名
                updateList(messageline.getBody());
            else if(messageline.getType().equals("MESSAGE"))//收到聊天信息
                message.append(messageline.getBody()+"\n"); //添加聊天信息
            else if(messageline.getType().equals("REMOVE")) //如果有用户离开
                message.append(messageline.getBody()+"离开聊天室\n");
        }
        else
            message.append("出现问题："+line+"\n");
    }
    /*更新在线用户名*/
    public void updateList(String line){
        users.setText("");
        String  str=line;
        for(int i=0;i<10;i++)
            userList[i]="";              //清除原有用户名
        int index=str.indexOf(":");      //获取第一个分号所在的位置
        int a=0;
        while(index!=-1){
            userList[a]=str.substring(0,index);  //用户名为第一个分号之前的字符串
            str=str.substring(index+1); //将第一个分号之后的字符串保存到 str 中
            a++;
            index=str.indexOf(":");      //获取新字符串中第一个分号所在的位置
        }
        for(int i=0;i<10;i++)            //输出在线用户名
            users.append(userList[i]+"\n");
    }
    /*读取服务器发送来的信息*/
    public void run(){
        try{
            String line="";
            while(connected&&line!=null){
                line=in.readLine();
                if(line!=null) extractMessage(line);
            }
        }catch(IOException e){}
    }
}
```

在这个程序中，限定同时在线聊天的用户最多不超过 10 个。

26.4.4 消息封装：Message 类

为了方便地判断服务器和客户端发送的信息的类别，在这里创建一个 Message 类对信息进行处理。Message 类的作用是将信息截取分为两部分，分号之前的部分为信息的类别，这部分信息是通过程序代码加入的，它只用来进行信息识别，而不会显示在聊天信息中。分号之后的部分为实际要显示的内容，具体的设置如下。

- USER：表示其分号后的信息为申请连接的客户端，也就是申请加入聊天室的用户。
- MESSAGE：表示其分号后的信息为服务器或客户端发送的信息。
- LIST：表示其分号后的信息为当前连接的客户端，也就是当前在线聊天的用户名。



- JOIN：表示其分号后的信息为新连接的客户端，也就是新加入到聊天室的用户名。
- LEAVE：表示其分号后的信息为与服务器断开的客户端，也就是离开聊天室的用户。
- REMOVE：表示其分号后的信息为需要从数组 chatters 中除去的客户端。

此外，“收到”表示服务器收到的信息，“发布信息”表示服务器向所有客户端发出的信息。

```
/*Message 类用来对信息进行处理*/
public class Message {
    private String type;                    //保存信息的类型，分号之前的部分
    private String body;                    //保存信息的具体内容，分号之后的部分
    private boolean valid;                  //信息是否有效，也就是信息是否含有：
    public Message(String messageLine){
        valid=false;
        type=body=null;
        int pos=messageLine.indexOf(":");
        if(pos>=0){
            type=messageLine.substring(0,pos).toUpperCase();
                                                    //分号之前的字符串
            body=messageLine.substring(pos+1); //分号之后的字符串
            valid=true;                             //信息有效
        }
    }
    public Message(String type,String body){
        valid=true;
        this.type=type;
        this.body=body;
    }
    /*返回信息的类型*/
    public String getType(){
        return type;
    }
    /*返回信息的具体内容*/
    public String getBody(){
        return body;
    }
    /*返回信息是否有效*/
    public boolean isValid(){
        return valid;
    }
}
```

从上面的程序中我们可以看到 socket 的使用过程。读者可以在单机上试验该程序，最好是能在真正的网络环境下试验该程序，这样更容易分辨输出的内容与客户机、服务器的对应关系。同时也可以修改该程序，提供更为强大的功能或更加满足读者的意图。

26.5　软件部署

用 Java 编写的程序一般都打包成 jar 文件，因为这样很方便用户使用。首先确保写的程序都有一个共同的包，也就是每一个 java 文件第一行都要有这一句：“package packagename;”，在此用的是“package net;”。因为心心聊天室是一个简单的 C/S 结构的 Java 程序，所以要把服务器端程序和客户端程序分别打包。下面对服务器端和客户端程序的打

包部署分别进行讲解。

26.5.1 服务器端程序打包与部署

服务器端程序打包与部署步骤如下所示。

① 使用“javac ChatServer.java”命令，获得所有与 ChatServer 类相关的.class 文件，在本案例中与 ChatServer 类相关的.class 文件为 ChatServer.class、ChatHandler.class 和 Message.class，然后把这些.class 文件放在文件夹 ChatS 内的 net 文件夹中。

② 在文件夹 ChatS 内，使用记事本创建一个纯文本文件，命名为 MANIFEST.MF。编辑 MANIFEST.MF 文件如下。

```
Manifest-Version: 1.0
Created-By: 1.6.0_04 (Sun Microsystems Inc.)
Main-Class: net.ChatS
```

创建 MANIFEST.MF 文件一定要注意：每一行的冒号和后面的内容之间要有空格；每一行的结尾一定要用回车键换行。其中 Main-Class：后面一定要跟带 main 方法的类的类名。

③ 在同一目录下，使用如图 26.5 所示的命令打包文件。

注意，jar 命令中，最后的圆点代表将当前目录的所有文件打包成 ChatS.jar 文件（文件名可任意取）。运行此命令之后，在当前目录下，会发现出现了一个名为 ChatS.jar 的文件，如图 26.6 所示。

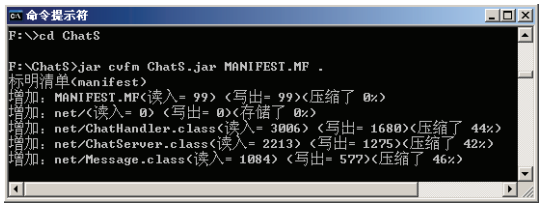


图 26.5 执行打包命令效果图

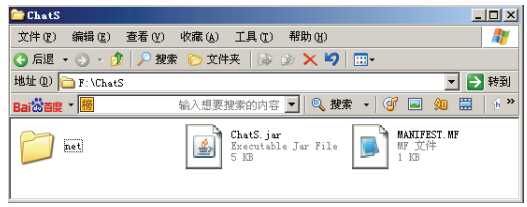


图 26.6 生成 ChatS.jar 文件效果图

26.5.2 客户端程序打包与部署

客户端程序打包与部署步骤如下所示。

① 使用“javac ChatClient.java”命令，获得所有与 ChatClient 类相关的.class 文件，在本案例中与 ChatClient 类相关的.class 文件为 ChatClient.class 和 Message.class，然后把这些.class 文件放在文件夹 ChatC 内的 net 文件夹中。

② 在文件夹 ChatC 内，使用记事本创建一个纯文本文件，命名为 MANIFEST.MF。编辑 MANIFEST.MF 文件如下。

```
Manifest-Version: 1.0
Created-By: 1.6.0_04 (Sun Microsystems Inc.)
Main-Class: net.ChatC
```

创建 MANIFEST.MF 文件一定要注意：每一行的冒号和后面的内容之间要有空格；每一行的结尾一定要用回车键换行。其中 Main-Class：后面一定要跟带 main()方法的类的类名。

③ 在同一目录下，使用如图 26.7 所示的命令打包文件：

```
Jar cvfm ChatC.jar MANIFEST.MF.
```

注意，最后的圆点代表将当前目录的所有文件打包成 ChatC.jar 文件（文件名可任意取）。运行此命令之后，在当前目录下，会发现出现了一个名为 ChatC.jar 的文件，如图 26-8 所示。



图 26.7　执行打包命令效果图

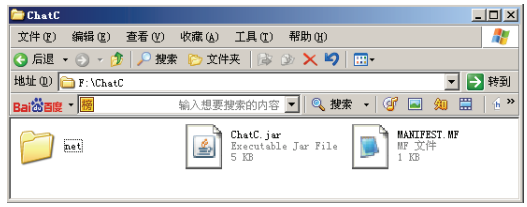


图 26.8　生成 ChatC.jar 文件效果图

26.5.3　运行测试

在服务器端和客户端程序到打包就绪后，就可以运行测试看看效果。测试分为两个步骤，首先要启动服务器程序，然后运行客户端程序进入主界面。

① 在文件夹 ChatS 中双击【ChatS.jar】文件，启动服务器端程序。

② 在文件夹 ChatC 中双击【ChatC.jar】文件，就可直接运行了，结果如图 26.9 所示。



图 26.9　运行 ChatC.jar 后的效果图

读者可以试着打开两个客户端，分别以不同的用户名登录，然后相互进行通信，互相发送聊天信息，以感受一下新编写的网络聊天室有趣而实用的功能吧。

26.6　小结

在本章中，通过一个完整的网络聊天室的开发过程，介绍了创建 Java 网络应用程序的步骤和方法。重点对程序的设计和功能模块划分及代码实现做了详细的说明。在本例中，综合运用到了 Java 的 Socket 编程技术、多线程处理等知识点，其中的多线程实现双向通信是本章学习的难点，读者可参考书中提供的示例源代码，实际上机调试并运行，以帮助理解。

下一章将介绍一个比较完整而复杂的 MIS 系统（管理信息系统），这也是实际开发中经常遇到的一类应用程序。

第27章　案例3：惠文教育教务管理系统

随着计算机技术的飞速发展和计算机在管理方面应用的普及，利用计算机来辅助办公、辅助管理势在必行。本章通过一个 IT 职业教育机构的教务管理系统的开发，综合利用学习过的 Java Swing 技术和数据库开发技术，使读者对小型 MIS 系统的开发过程有深刻的认识，并提升综合运用 Java 技术的能力。



27.1　背景与主要技术介绍

教务管理系统实现的是学校（一般为教务科或教务处）对学校教务工作的信息化管理。使用教务管理系统，能够提高学校的管理水平，提高教学事务处理的效率，节省人力和时间成本。本节就介绍一下本项目的应用背景和涉及的主要技术。

27.1.1　应用背景

惠文教育是一所从事 IT 高端职业教育的专业教育机构，已有 6 年的发展历史，近年来随着信息技术的飞速发展，惠文教育的规模也快速扩大。原先，惠文教育的教务工作只需要两个教务专员即可，但是今年以来，教务部门的教务专员数量已经增加到 8 个。学生数量增长也很快，已由几百人增加到了目前的在校生近三千人。随着规模的扩大，教务部门的工作也日趋繁重，因此，惠文教育决定开发一个教务管理软件，作为校园电子办公系统的重要组成部分，代替一直以来手工统计、纸质文档管理的工作方式，并且将学生的学籍管理、教学计划制定、教学资源的分配及教务相关工作，统统迁移到软件上来。各个部门也能共享教务信息，如学生信息、员工信息、考试成绩等。

下面就一步一步介绍教务系统从总体设计、数据库设计到最后系统实现的整个开发过程。

27.1.2　安装和配置 MySQL 数据库

教务管理系统中涉及大量的数据信息，如学生学籍信息、教职员工信息等，因此，必



须有一个可靠而稳定的数据库来管理数据。本系统采用的是 MySQL 数据库。MySQL 是一个非常轻巧但功能强大的数据库，安装容易使用简单，并且支持多用户多线程、网络连接、并发访问及 JDBC，更重要的是，MySQL 数据库使用起来免费，很适合于学校使用。下面介绍如何下载、安装、配置及创建数据库。

① 下载 MySQL 数据库。打开 IE 浏览器，在地址栏中输入网址“http://www.mysql.com/”，访问 MySQL 的网站，找到下载页面。或者直接在地址栏输入“http://dev.mysql.com/downloads/mysql/5.1.html”，下载 MySQL 最新的 Windows 版本，如图 27.1 所示。

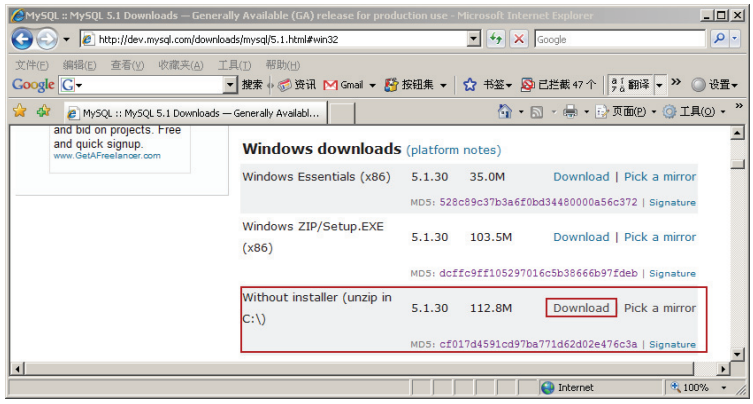


图 27.1　下载 MySQL 数据库

需要下载：

- MySQL 数据库免安装版（不需要安装，解压缩后配置系统变量即可）。
- MySQL 的 JDBC 驱动程序。

② 安装 MySQL 数据库。安装 MySQL 数据库非常容易，只需要将下载的免安装的 ZIP 文件解压缩到 C 盘即可。在这里，将其安装目录设为：C:\MySQL51，如图 27.2 所示。

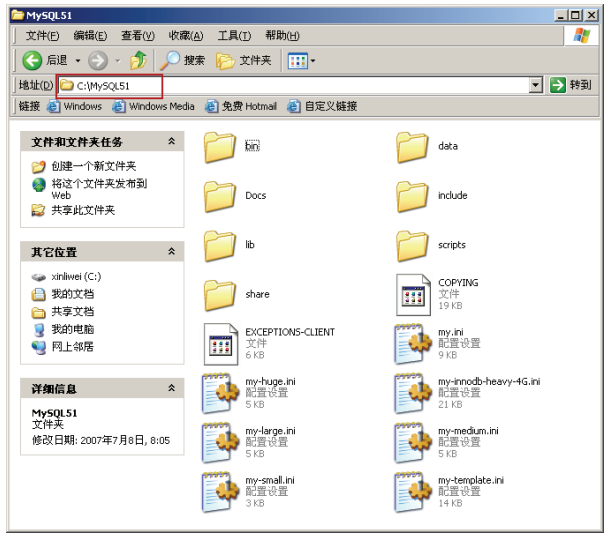


图 27.2　MySQL 安装程序

③ 配置环境变量。要使用 MySQL 数据库，首先要做一些环境设置。同配置 JDK 的

环境变量一样，打开【编辑系统变量】对话框，将 MySQL 的安装路径（例如，C:\MySQL51\bin，依实际安装的路径设置）加入 path 变量即可，如图 27.3 所示。设置好环境变量之后，就可以在命令行下任何地方使用 MySQL 数据库命令了。

④ 启动 MySQL 服务器。首先要启动 MySQL 数据库服务器，才能从客户端进行数据库访问。选择【开始】|【运行】命令，打开【运行】对话框，如图 27.4 所示，输入“cmd”命令，单击【确定】按钮，打开【命令提示符】窗口。

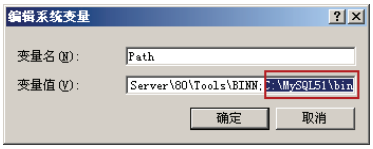


图 27.3　编辑系统变量



图 27.4　进入命令行窗口

在【命令提示符】窗口下，输入“mysqld-nt”命令，并按 Enter 键，启动 MySQL 数据库服务器，如图 27.5 所示。



图 27.5　启动 MySQL 服务器

⑤ 启动 MySQL 数据库客户端程序。另打开一个【命令提示符】窗口，输入“mysqld –u root –p”命令，并按 Enter 键，此时系统提示输入密码。MySQL 安装后，有一个初始用户账号 root，初始密码为空，因此不需输入密码，直接按 Enter 键，启动 MySQL 数据库客户端程序，如图 27.6 所示。

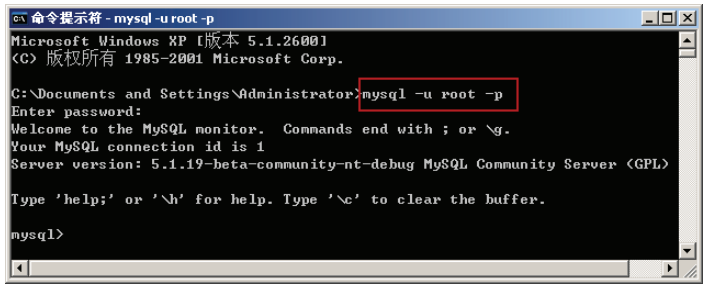


图 27.6　启动 MySQL 客户端程序

⑥ MySQL 数据库安全设置。为了 MySQL 的安全，一般要求用户为 root 账号重新设置密码。在命令行窗口中，使用如下命令为 root 账号设置新密码：

```
--进入 mysql 数据库
use mysql
--为 root 账号重新设置密码，新的密码为:1234
update user set password=PASSWORD('1234') where user='root';
flush privileges;
```

```
--退出 mysql 客户程序
quit
```

在使用时，一般不使用 root 账号登录，而是另外创建用户账号和密码。假设这里创建一个新的账号：dbuser，密码为：1234。再以 root 账号进入 MySQL 客户程序，命令如下。

```
mysql -u root -p
```

其中-u 后面跟的是账号，-p 指示后面的密码。这时系统会提示输入密码，输入：1234，然后再进入 mysql 数据库，创建新的账号 dbuser：

```
--进入 mysql 数据库
use mysql;
--创建新的账号和密码
grant all privileges on *.* to dbuser@localhost identified by '1234' with
grant option;
--退出 mysql 客户程序
quit
```

⑦ 创建数据库 accp，并在 accp 中创建数据表 departmentinfo。同样，在命令行窗口下，使用如下命令创建（注意，以‘--’开头的为 MySQL 中的注释，‘/**　*/’也表示注释）：

```
--以 dbuser 账号进入 mysql 客户程序
mysql -u dbuser -p
--根据系统提示，输入密码：
1234
--创建数据库 accp
create database accp;
--进入数据库 accp
use accp;
--在 accp 数据库中,创建表示部门信息的数据表 departmentinfo
create table DEPARTMENTINFO(
    DEPARTMENTID CHAR(4) not null primary key, /** 部门编号，主键  */
    DEPARTMENTNAME VARCHAR(15) not null        /** 部门名称，不能为空  */
);
--在 departmentinfo 表中加入部门记录
insert into DEPARTMENTINFO values ('hw01', '行政部');
insert into DEPARTMENTINFO values ('hw02', '市场部');
insert into DEPARTMENTINFO values ('hw03', '教质部');
insert into DEPARTMENTINFO values ('hw04', '学术部');
insert into DEPARTMENTINFO values ('hw05', '就业部');
insert into DEPARTMENTINFO values ('hw06', '财务部');
insert into DEPARTMENTINFO values ('hw07', '后勤部');
insert into DEPARTMENTINFO values ('hw08', '人事部');
--退出 mysql
quit
```

⑧ 查看数据库信息。如果想查看数据库的信息，使用如下的命令：

```
-以 dbuser 账号进入 mysql 客户程序
mysql -u dbuser -p
--根据系统提示，输入密码：
1234
--查看系统中数据库的信息
show databases;
```

会列出系统中存在的所有数据库的名称，如图 27.7 所示。

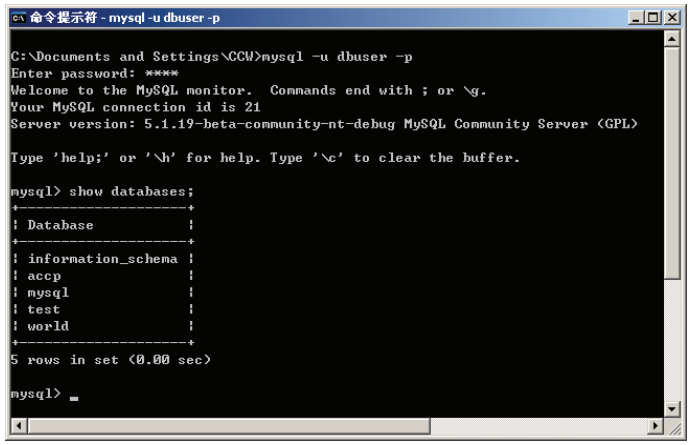


图 27.7　启动 MySQL 并查看数据库信息

⑨ 查看 accp 中的数据表信息。如果想查看 accp 数据库中的数据表信息，使用如下的命令：

```
--进入数据库 accp
use accp;
--查看 accp 数据库中表的信息
show tables;
```

会列出 accp 数据库中所有的数据表，如图 27.8 所示。

⑩ 查看数据表结构。如果想查看 accp 数据库中 departmentinfo 数据表的结构信息，使用如下的命令：

```
--查看 accp 数据库中 departmentinfo 表的结构
desc departmentinfo;
```

或者

```
--查看 accp 数据库中 departmentinfo 表的结构
describe departmentinfo;
```

会列出 accp 数据库中 departmentinfo 表的数据结构，如图 27.9 所示。

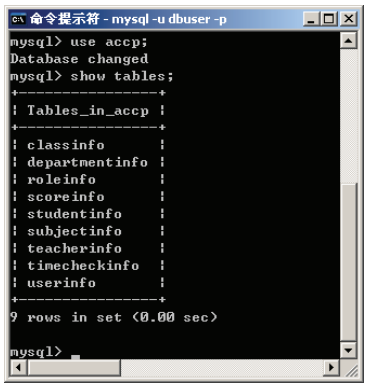


图 27.8　查看 accp 数据库中数据表

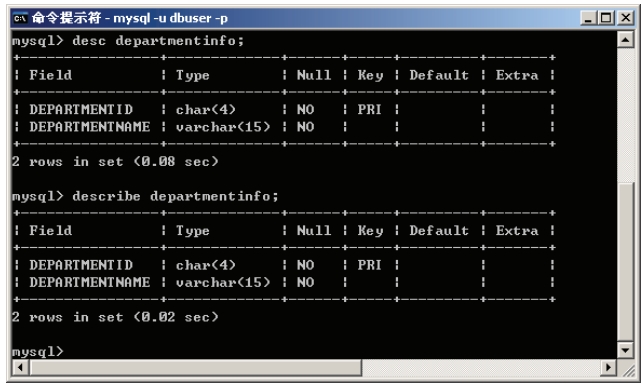


图 27.9　查看 departmentinfo 表的数据结构



⑪ 在启动 MySQL 后，输入“mysql”命令，并按 Enter 键，可以看到 MySQL 可以使用的命令，如图 27.10 所示。

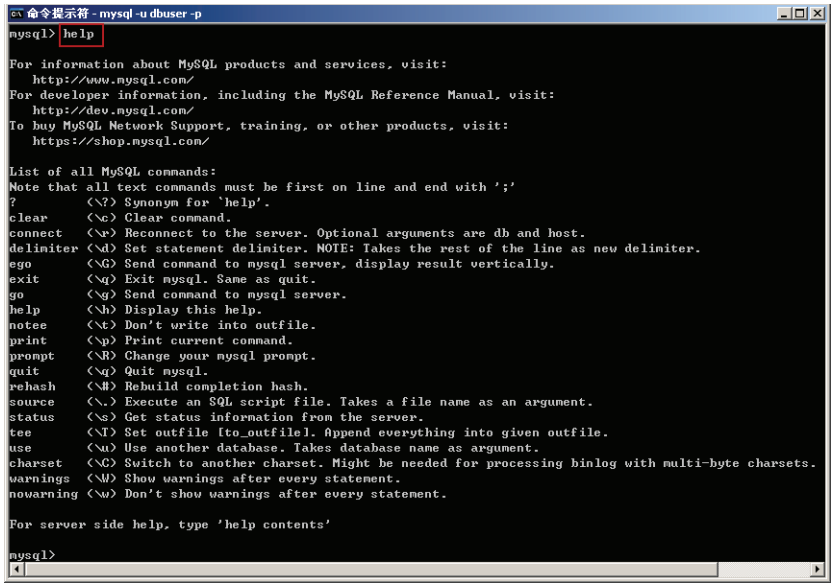


图 27.10　MySQL 可以使用的命令

27.1.3　配置 MySQL 数据库的 jdbc 驱动程序

在数据库应用程序中，需要在程序中加载 JDBC 驱动程序，以访问数据库。所以在进行开发之前，需要配置好 MySQL 的 JDBC 驱动程序路径。

考虑到可移植性，将下载的 MySQL 驱动程序的 jar 包（本书中使用 mysql-connector-java-5.1.0-bin.jar），解压缩到应用程序的文件夹下（如“C:\examples\chapter27\”文件夹下）。解压缩以后有一个 com 文件件，要用到的 MySQL 驱动程序的类“com.mysql.jdbc.Driver”就在其中。另外，在本程序中，还使用到了一个名为“com.borland.jbcl.layout.XYLayout”的布局管理器类，也一同位于 com 文件夹下，如图 27.11 和图 27.12 所示。

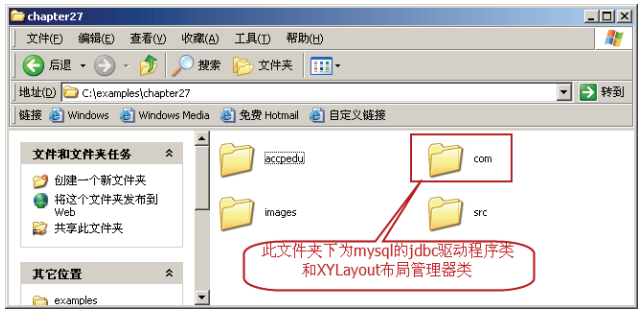


图 27.11　配置 MySQL 数据库驱动程序

这样配置以后，在数据库应用程序中，就可以使用 MySQL 的驱动程序类和 XYLayout 布局管理器类了。

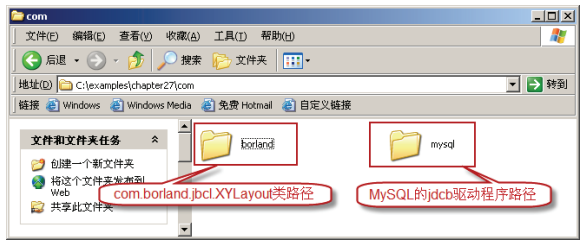


图 27.12　com 包

27.2　总体设计与概要说明

经过对惠文教育教务处教务工作的深入了解，经过与教务人员的沟通，在对系统需求进行分析的基础上，对惠文教育教务管理系统的总体功能模块的设计如下。

27.2.1　功能模块划分

经过了解，惠文教育的教务管理系统的使用的人员（角色），主要有三类，分别为班主任、教务专员、人事专员。不同角色对教务管理系统的使用如表 27.1 所示。

表 27.1　角色与功能模块

角　色	操　作
班主任	负责学生的考勤管理和档案管理
教务专员	负责班级信息管理（新开班级信息、修改班级信息），并能对错误班级信息进行删除操作
人事专员	负责人事信息管理（员工入职、离职、部门调整等）

根据不同的角色及其需进行的操作，将教务管理系统总体模块设计如图 27.13 所示。

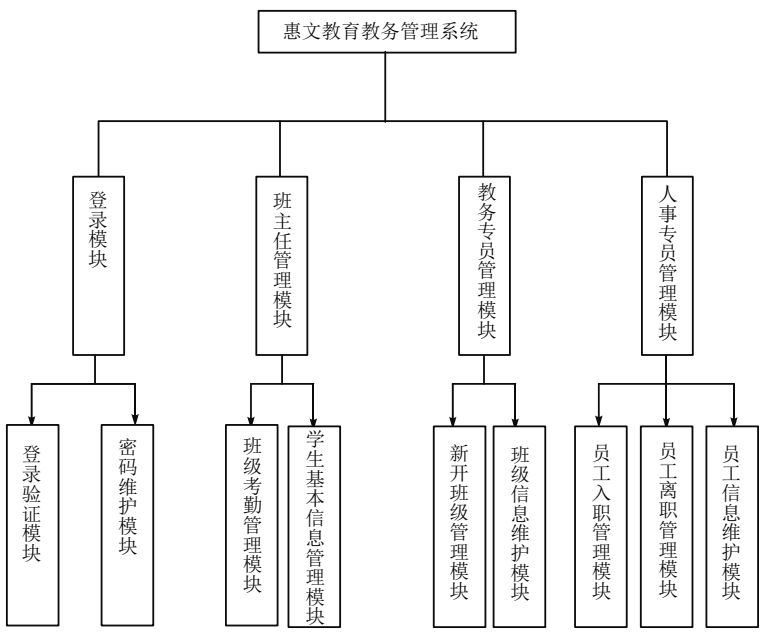


图 27.13　系统总体模块设计

27.2.2　功能模块说明

根据系统总体模块设计，对其中的每个模块和其子模块的功能简要说明如下。

1．登录模块

登录模块提供登录账号、密码及登录角色验证。用户名为统一设定的员工编号，如“301A008”，具有唯一性。密码要求系统为所有账号提供统一的初始密码，如“111111”，用户登录以后可以自行修改密码。用户如果以初始密码（“111111”）登录，系统要提醒用户及时更改密码。修改密码时，需要两次确认。在登录时，要选择要登录的角色。不同的角色，系统提供不同的权限。以不同的角色登录，系统自动出现与所登录角色权限相适应的操作界面。

2．班主任管理模块

班主任可以对其所带班级中指定的班级进行考勤，包括对班级中每个人某天出勤情况的录入和考勤查询。学生出勤记录分为：出勤、迟到、请假和旷课。班主任负责管理其所带班级的学生基本信息的管理，包括新入学学生档案信息的录入、在校学生档案信息的修改，以及对非毕业离校学生信息的删除（如开除、劝退等）。

3．教务专员管理模块

教务专员负责对新开班的班级信息进行管理，如为新开班级指定班级编号、班主任和授课教师等。教务专员还负责对班级信息进行维护，如任课教师变更、带班班主任变更、学生休学或转学等。

4．人事专员管理模块

人事专员负责学校人事管理，包括浏览学校所有部门的人员信息或各部门的人员信息、新员工入职管理、员工离职管理、员工工作部门变更等。

27.3　业务流程图

程序员必须对应用程序要处理的业务流程非常清晰，才能正确地编写业务逻辑代码。本节对惠文教育教务管理的业务流程进行分析。

27.3.1　登录模块流程

登录操作比较简单，账号和密码及每个账号所分配的角色都存储在数据库中，系统在接到账号、密码及对应的角色选择后，与数据库建立连接，然后在数据库中查找是否有所接到的账号、密码和角色存在，进行验证。如果存在的话，就返回正确的信息，并根据登录的角色，进入相应的管理子系统。如果验证不正确的话，必须给用户以提示信息，说明哪里不正确（如账号和密码不正确，或角色选择与对应账号所分配的角色不匹配）。

另外，应用程序要健壮，那么一定要进行出错处理，例如在连接数据库失败时，要输出出错信息让用户知道问题出在哪里。

登录模块的业务流程如图 27.14 所示。

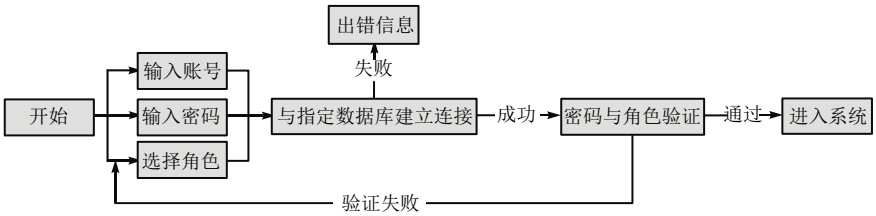


图 27.14　登录模块流程图

27.3.2　班主任管理模块流程

班主任登录模块相对稍微复杂一点。班主任进入系统以后，可以对要进行的子操作进行选择，包括浏览所带班级（有可能一个班主任带多个班级）、对指定的班级进行考勤管理、维护指定班级学生的基本信息。这些功能的业务流程如图 27.15 所示。

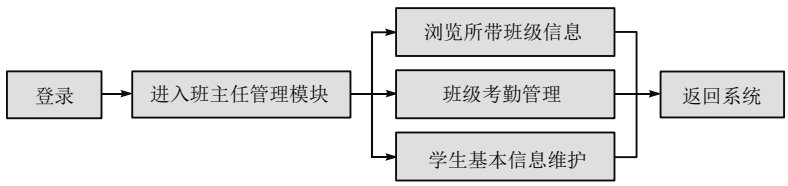


图 27.15　班主任管理模块流程图

27.3.3　教务专员管理模块

教务专员进入系统以后，可以浏览和维护班级信息，对新开班级进行管理。其业务流程如图 27.16 所示。

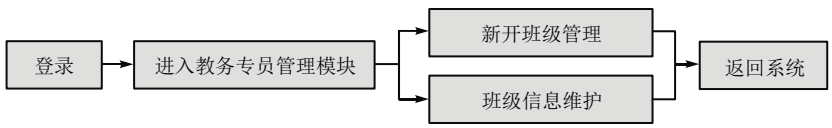


图 27.16　教务专员管理模块流程图

27.3.4　人事专员管理模块

人事专员进入系统以后，可以对以下业务模块进行操作：新入职员工管理、离职员工管理、员工信息维护管理。其业务流程图如图 27.17 所示。

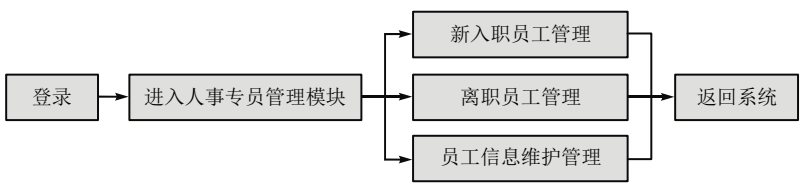
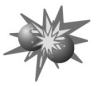


图 27.17　人事专员管理模块流程图

27.4　数据库设计

教务管理系统是一个数据库应用系统，教学教务等所有信息都保存在数据库中。在数



据库应用系统的开发过程中，数据库的结构设计是一个非常重要的问题。这里所说的数据库结构设计是指数据库中各个表结构的设计，包括信息保存在哪些表格中、各个表格的结构如何，以及各个表之间的关系。

本节开始介绍的是本项目的数据库实现细节，包括数据库的需求分析、数据库结构设计、数据库逻辑结构设计和数据库结构的实现。

27.4.1　数据库需求分析

数据库结构设计的好坏，将直接影响到应用系统的效率及实现的效果。好的数据库结构设计会减少数据库的存储量，数据的完整性和一致性就高，系统会有较快的响应速度，并有助于简化基于此数据库的应用程序的实现等。而设计数据库的第一步，是数据库需求分析。

在数据库需求分析阶段，主要是分析基本数据及数据处理的流程，为进一步的设计打下基础。在惠文教务管理系统中，通过对各个功能模块的内容和过程进行分析，所设计的数据项和数据结构如下。

- 用户。包括的数据项有：用户账号、密码和角色类型。
- 班级基本信息。包括的数据项有班级编号、总部班级信息、授课教员、班主任、开班日期、预计结课日期、预计参加考试时间、班级种类、周课时数、开课人数、转入、转出、休学、班级人数、结课日期、结课时人数、预计升学率、获证人数、实际开学人数、实际升学率、学期。
- 学生考勤信息。包括的数据项有：学生学号、出勤状态、考勤时间。
- 学生基本信息。包括的数据项有：中心编号、学员学号、学员姓名、姓名拼音、性别、身份证号、所在班级编号、班主任编号、学员状态、已持有证书、学历、专业、毕业学校、工作状态、工作单位、工作种类、联系电话、通讯地址、邮编、就业意向城市、就业意向单位、备注。
- 单位的部门信息。包括的数据项有：部门编号、部门名称。
- 角色信息。包括的数据项有：角色编号、角色名称。
- 教师基本信息。包括的数据项有：员工编号、员工姓名、出生日期、性别、毕业院校、专业、籍贯、入职时间、所在部门、职务。

总结了上面的数据结构和数据项之后，就可以进行下面的数据库设计了。

27.4.2　数据库概念结构设计

数据库概念结构设计阶段，是在需求分析的基础上，设计出能够满足用户需求的各种实体，以及它们之间的关系，为后面的逻辑结构设计打下基础。

这个阶段不需要考虑所采用的数据库管理系统、操作系统和机器类型等问题。在这个阶段，一般使用 E-R 图（Entity-Relation，实体-联系图）进行数据库概念设计。E-R 图是描述数据表及其关系的一种直观的描述工具，图中包括：

- 实体。用方框表示，方框内为实体的名称。
- 实体的各种属性。用椭圆来表示，椭圆内为属性名称。使用线段将其和相应的实体联系起来。

- 实体之间的联系。用菱形表示，菱形内为联系的名称。实体和实体之间的联系可分为 1 对 1（1:1）、1 对多（1:*n*）和多对多（*m*:*n*）三种。

教务管理系统对数据库系统的要求是：

- 结构合理。
- 所建立的数据冗余度要小，独立性要强。
- 增加、修改、查询和统计快速而准确。
- 保密性和可靠性要好。

根据需求分析，设计本教务管理系统的实体的 E-R 图描述如图 27.18～27.23 所示。

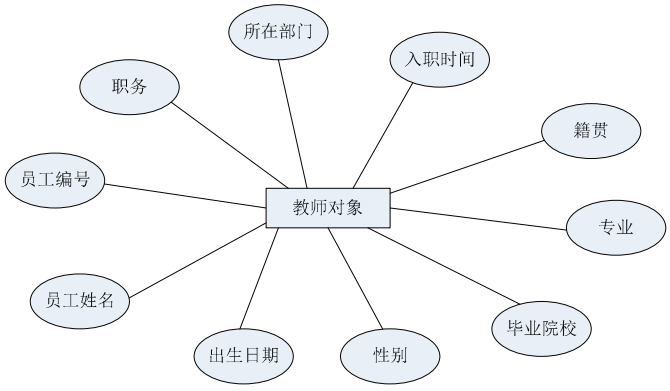


图 27.18　教师对象实体 E-R 图

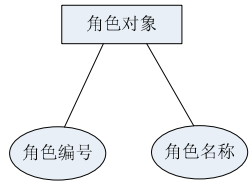


图 27.19　角色对象实体 E-R 图

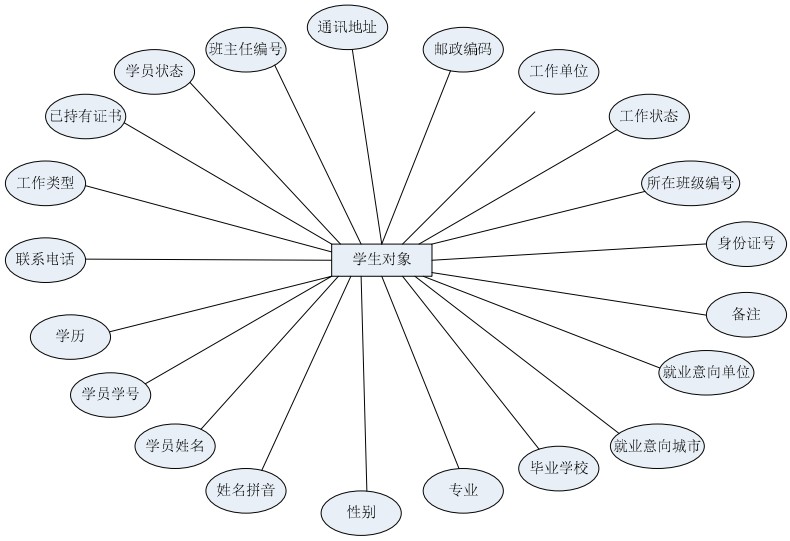


图 27.20　学生对象实体 E-R 图

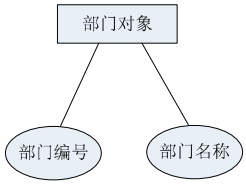


图 27.21　部门对象实体 E-R 图

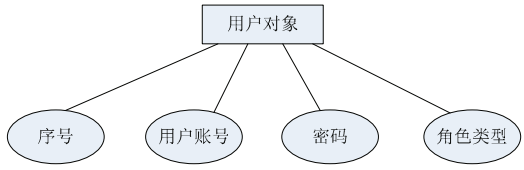


图 27.22　用户对象实体 E-R 图

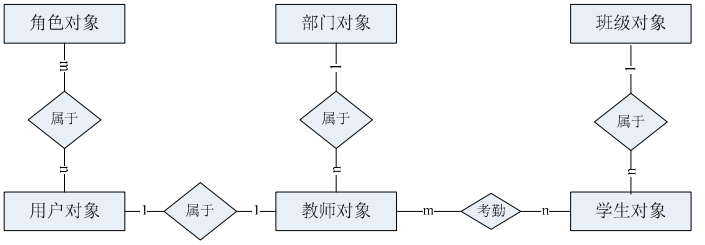


图 27.23　实体之间的关系 E-R 图

其中图 27.23 为实体之间的关系 E-R 图。有了 E-R 图之后，就可以根据 E-R 图将其转化为逻辑结构了。

27.4.3　数据库逻辑结构设计

概念结构是独立于实际数据模型的信息结构，必须将其转化为逻辑结构后才能进行数据库应用的设计。也就是要将概念上的结构转化为数据库系统所支持的实际数据模型。在本项目中采用的是 MySQL 数据库管理系统。

第一种概念结构向逻辑结构的转化是将实体转化为关系表。这种转化比较简单，只需要将实体的属性定义为表的属性即可。

第二种转化是联系的转化。即将各个实体之间的联系转化为表格之间的关系，如外键的定义，或者定义单独的关系表。

在上面工作的基础上，归纳出教务管理系统数据库表格的组成、列的属性以及表格之间的联系等。教务管理系统共有 7 张表，其逻辑结构如表 27.2～27.8 所示。

表 27.2 为部门基本信息表 DEPARTMENTINFO 表。

表 27.2　DEPARTMENTINFO（角色信息表）

列　　名	数据类型	字　　长	可否为空	是否主键	默 认 值	说　　明
DEPARTMENTID	char	4	NO	PRI		
DEPARTMENTNAME	varchar	15	NO			

表 27.3 为教师基本信息表 TEACHERINFO 表。

表 27.3　TEACHERINFO（角色信息表）

列　　名	数据类型	字　　长	可否为空	是否主键	默 认 值	说　　明
teacherid	char	15	NO	PRI		
teachername	varchar	10	NO			
teacherbirthday	varchar	20	YES		NULL	
teachersex	enum('男','女')		NO		男	
teacherschool	varchar	20	YES		NULL	
teachertech	varchar	20	YES		NULL	
teacheraddress	varchar	20	YES		NULL	
teacherindate	varchar	20	YES		NULL	
teacherdepartment	varchar	12	YES	MUL	NULL	
teacherduty	varchar	16	YES		NULL	

表 27.4 为角色表 ROLEINFO 表。

表 27.4　ROLEINFO（角色信息表）

列　　名	数据类型	字　　长	可否为空	是否主键	默 认 值	说　　明
roleid	char	1	NO	PRI		
rolename	varchar	20	YES		NULL	

表 27.5 为用户信息表 USERINFO 表。

表 27.5　USERINFO（角色信息表）

列　　名	数据类型	字　　长	可否为空	是否主键	默 认 值	说　　明
userno	Int	11	NO	PRI	NULL	auto_increment
userid	char	15	NO			
userpwd	varchar	20	NO		111111	
userrole	varchar	6	YES		NULL	

表 27.6 为班级信息表 CLASSINFO 表。

表 27.6　CLASSINFO（角色信息表）

列　　名	数据类型	字　　长	可否为空	是否主键	默 认 值	说　　明
CLASSID	char	15	NO	PRI		
CLASSBID	char	10	NO			
CLASSTEACHER	varchar	10	YES		NULL	
CLASSMANAGER	varchar	10	YES		NULL	
CLASSBEGINDATE	varchar	10	YES		NULL	
CLASSFINISHDATE	varchar	10	YES		NULL	
CLASSTESTDATE	varchar	10	YES		NULL	
CLASSTYPE	varchar	10	YES		NULL	
CLASSPERWEEK	smallint	6	YES		12	
CLASSBEGINNUMBER	smallint	6	YES		0	
CLASSSTUIN	smallint	6	YES		0	
CLASSSTUOUT	smallint	6	YES		0	
CLASSSTUPAUSE	smallint	6	YES		0	
CLASSSTUNUMBER	smallint	6	YES		0	
CLASSENDDATE	varchar	10	YES		NULL	
CLASSENDNUMBER	smallint	6	YES		0	
CLASSPRERATE	float		YES		0	
CLASSPASSNUMBER	smallint	6	YES		0	
CLASSREALNUMBER	smallint	6	YES		0	
CLASSREALRATE	float		YES		0	
CLASSSEME	char	2	YES		S1	

表 27.7 为学生基本信息表 STUDENTINFO 表。

表 27.7　STUDENTINFO（角色信息表）

列　　名	数据类型	字　　长	可否为空	是否主键	默 认 值	说　　明
schooled	char	3	NO		301	
stuid	char	11	NO	PRI		
stuname	varchar	10	NO			
stunamespell	varchar	20	NO			



续表

列　名	数据类型	字　长	可否为空	是否主键	默 认 值	说　明
stusex	enum('男','女')		NO		男	
stuidentity	char	18	YES		NULL	
stuclassid	char	15	YES		NULL	
stuteacherid	char	15	YES		NULL	
stustate	varchar	10	YES		在读	
stucertified	varchar	20	YES		NULL	
stuedulevel	varchar	10	NO		高中	
stusped	varchar	20	YES		NULL	
stuhomeschool	varchar	20	YES		NULL	
stuworkstate	varchar	20	NO		待业	
stucompany	varchar	10	YES		NULL	
stuworktype	varchar	20	YES		NULL	
stutelphone	varchar	20	YES		NULL	
stuaddress	varchar	20	YES		NULL	
stucode	char	6	YES		NULL	
stujobcity	varchar	15	YES		NULL	
stujobcompany	varchar	20	YES		NULL	
remark	varchar	250	YES		NULL	

表 27.8 为考勤表 TIMECHECKINFO 表。

表 27.8　TIMECHECKINFO（角色信息表）

列　名	数据类型	字　长	可否为空	是否主键	默 认 值	说　明
stuid	char	11	YES		NULL	
checkedstate	enum('出勤','迟到','请假','旷课','早退')	20	NO		出勤	
checkedtime	varchar	10	YES		NULL	

27.4.4　数据库结构的实现

在需求分析和概念结构设计的基础上得到数据库的逻辑结构之后，就可以在 MySQL 数据库系统中实现该逻辑结构。首先启动 MySQL 服务器，然后启动 MySQL 客户端程序，按以下步骤实现数据库结构：

① 以 dbuser 身份，密码 1234 启动 MySQL 客户端程序：

```
mysql -u dbuser -p
Enter password: ****
```

② 创建数据库 accp：

```
create database accp;
```

③ 打开数据库 accp：

```
use accp;
```

④ 创建部门基本信息表（DEPARTMENTINFO）：

```
create table DEPARTMENTINFO(
    DEPARTMENTID CHAR(4) not null primary key,     /** 部门编号  */
    DEPARTMENTNAME VARCHAR(15) not null            /** 部门名称  */
```

```
);
```

⑤ 创建教师基本信息表（TEACHERINFO）：

```
create table TEACHERINFO3(
    teacherid CHAR(15) not null primary key,       /** 员工编号  */
    teachername VARCHAR(10) not null,              /**  员工姓名  */
    teacherbirthday VARCHAR(20),                   /**  出生日期  */
    teachersex ENUM("男","女") not null default '男',      /**  性别  */
    teacherschool VARCHAR(20),                     /**  毕业院校  */
    teachertech VARCHAR(20),                       /**  专业  */
    teacheraddress VARCHAR(20),                    /**  籍贯  */
    teacherindate VARCHAR(20),                     /**  入职时间  */
    teacherdepartment CHAR(4),     /**  所在部门:外键表：departmentinfo  */
    teacherduty VARCHAR(16),       /**  职务  */
    FOREIGN KEY(teacherdepartment) REFERENCES departmentinfo(DEPARTMENTID)
);
```

⑥ 创建角色表（ROLEINFO）：

```
create table ROLEINFO(
    roleid CHAR(1) not null primary key,           /**  角色编号  */
    rolename VARCHAR(20)                           /**  角色名称  */
);
```

⑦ 创建用户信息表（USERINFO）：

```
create table USERINFO(
    userno INTEGER not null auto_increment,        /**  序号  */
    userid CHAR(15) not null,                      /**  员工编号，同时作为用
户登录账号  */
    userpwd VARCHAR(20) not null default '111111', /**  账户密码  */
    userrole VARCHAR(6),                           /**  用户所分配的角色  */
    PRIMARY KEY  (userno)                          /**  主键  */
);
```

⑧ 创建班级信息表（CLASSINFO）：

```
CREATE TABLE CLASSINFO
(
    CLASSID CHAR(15) not null primary key,     /** 班级编号  */
    CLASSBID CHAR(10) not null,                /**  总部班级编号  */
    CLASSTEACHER VARCHAR(10),                  /**  授课教员  */
    CLASSMANAGER VARCHAR(10),                  /**  班主任  */
    CLASSBEGINDATE VARCHAR(10),                /**  开课日期  */
    CLASSFINISHDATE VARCHAR(10),               /**  预计结课日期  */
    CLASSTESTDATE VARCHAR(10),                 /**  预计参加考试时间  */
    CLASSTYPE VARCHAR(10),                     /**  班级种类  */
    CLASSPERWEEK SMALLINT default 12,          /**  周课时数  */
    CLASSBEGINNUMBER SMALLINT default 0,       /**  开课人数  */
    CLASSSTUIN SMALLINT default 0,             /**  转入  */
    CLASSSTUOUT SMALLINT default 0,            /**  转出  */
    CLASSSTUPAUSE SMALLINT default 0,          /**  休学  */
    CLASSSTUNUMBER SMALLINT default 0,         /**  班级人数  */
    CLASSENDDATE VARCHAR(10),                  /**  结课日期  */
    CLASSENDNUMBER SMALLINT default 0,         /**  结课时人数  */
```



```
    CLASSPRERATE FLOAT default 0.0,                  /**  预计升学率  */
    CLASSPASSNUMBER SMALLINT default 0,              /**  获证人数  */
    CLASSREALNUMBER SMALLINT default 0,              /**  实际开学人数  */
    CLASSREALRATE FLOAT default 0.0,                 /**  实际升学率  */
    CLASSSEME CHAR(2) default 'S1'                   /**  学期  */
);
```

⑨ 创建学生基本信息表（STUDENTINFO）：

```
create table STUDENTINFO(
    schoolid CHAR(3) not null default '301',         /**  中心编号  */
    stuid CHAR(11) not null primary key,             /**  学员学号  */
    stuname VARCHAR(10) not null,                    /**  学员姓名  */
    stunamespell VARCHAR(20) not null,               /**  姓名拼音  */
    stusex ENUM("男","女") not null default '男',     /**  性别  */
    stuidentity CHAR(18),                            /**  身份证号  */
    stuclassid CHAR(15),                             /**  所在班级编号  */
    stuteacherid  CHAR(15),                          /**  班主任编号  */
    stustate VARCHAR(10) null default '在读',        /**  学员状态  */
    stucertified VARCHAR(20),                        /**  已持有证书  */
    stuedulevel VARCHAR(10) not null default '高中', /**  学历  */
    stuspec VARCHAR(20),                             /**  专业  */
    stuhomeschool VARCHAR(20),                       /**  毕业学校  */
    stuworkstate VARCHAR(20) not null default '待业',/**  工作状态  */
    stucompany VARCHAR(10),                          /**  工作单位  */
    stuworktype VARCHAR(20),                         /**  工作种类  */
    stutelphone VARCHAR(20),                         /**  联系电话  */
    stuaddress VARCHAR(20) ,                         /**  通讯地址  */
    stucode CHAR(6),                                 /**  邮编  */
    stujobcity VARCHAR(15) ,                         /**  就业意向城市  */
    stujobcompany VARCHAR(20),                       /**  就业意向单位  */
    remark VARCHAR(250)                              /**  备注  */
);
```

⑩ 创建考勤表（TIMECHECKINFO）：

```
create table TIMECHECKINFO(
    stuid CHAR(11),                                  /**  学员学号  */
    checkedstate ENUM("出勤","迟到","请假","旷课","早退") not null default '
出勤',                                               /**  出勤状态  */
    checkedtime VARCHAR(10)                          /**  考勤时间  */
);
```



27.5　应用程序实现

上面创建了教务管理系统的数据库结构，数据库中的各个表都是空的。下面分小节来讲述如何在 Java 环境下完成教务管理系统的设计。

27.5.1　创建应用程序对象

首先建立一个名为 AccpApp.java 的主应用程序，含有 main()方法。代码如下所示。

```
package accpedu;
```

```
import java.awt.Toolkit;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;
import java.awt.Dimension;
public class AccpApp {
    boolean packFrame = false;
    //构造器，构造登录窗体
    public AccpApp() {
        LoginFrame frame = new LoginFrame();
        // 验证窗体是否以首选大小显示
        if (packFrame) {
            frame.pack();
        } else {
            frame.validate();
        }
        // 将窗口居中显示
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
                                                        //获取屏幕大小
        Dimension frameSize = frame.getSize();          //获取登录窗体大小
        if (frameSize.height > screenSize.height) { //如果登录窗体比屏幕高
            frameSize.height = screenSize.height; //将登录窗体高度设为屏幕高度
        }
        if (frameSize.width > screenSize.width) {   //如果登录窗体比屏幕宽
            frameSize.width = screenSize.width;  //将登录窗体宽度设为屏幕宽度
        }
        frame.setLocation((screenSize.width - frameSize.width) / 2,
                                                        //设置登录窗体居中显示
                          (screenSize.height - frameSize.height) / 2);
        frame.setVisible(true);                         //显示登录窗体
    }
    // 主方法，应用程序入口
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                try {                                   //将程序外观设为跨平台外观
                    //UIManager.setLookAndFeel(UIManager.
getSystemLookAndFeelClassName());
                    UIManager.setLookAndFeel(UIManager.
getCrossPlatformLookAndFeelClassName());
                } catch (Exception exception) {
                    exception.printStackTrace();
                }
                new AccpApp();                          //启动应用程序
            }
        });
    }
}
```

本教务系统的所有类均位于包 accpedu 中，应用程序 AccpApp 为主程序，含有应用程序入口 main()方法。在 AccpApp 的构造器中，构建登录窗体，设置登录窗口出现在屏幕的中央。



27.5.2　实现登录模块

由于教务管理系统安全性要求较强，因此所有用户首先要登录以后才能进行相应模块的操作。在应用程序启动以后，首先要调用的就是登录模块。本节将讲述系统登录窗口的设计与实现。

惠文教育教务管理系统的登录窗体设计如图 27.24 所示。



图 27.24　登录窗体的设计

功能设计：

- 如果账号或密码为空，则不能登录，并且给用户以提示。
- 登录用户必须选择一个角色，即必须以某个角色登录。否则不能登录，并且给用户以提示。
- 单击【登录】按钮，进行登录验证。如果通过，则进入相应角色的管理界面；如果不通过，给用户以提示。
- 单击【重置】按钮，则所有文本框清空，组合框恢复到未选择状态。
- 单击【退出系统】按钮，则程序结束，退出系统。

登录窗体类 LoginFrame 为 javax.swing.JFrame 的子类，其代码实现如下所示（其中由集成开发工具所自动生成的代码在此省略，读者可以查看本书光盘提供的相关源代码）：

```
package accpedu;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import com.borland.jbcl.layout.XYLayout;
import com.borland.jbcl.layout.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
public class LoginFrame extends JFrame {
    JPanel contentPane;                              //声明内容面板对象
    XYLayout xYLayout1 = new XYLayout();             //创建 XYLayout 对象
    XYLayout xYLayout2 = new XYLayout();
    JLabel jLabel1 = new JLabel();                   //声明显示“惠文教育”的标签
```

```
    //此处省略基本的控件声明代码，使用集成开发工具会自动生成
    ……

    JPanel jPanel1 = new JPanel();
    public LoginFrame() {                          //登录窗体类的构造器
        try {
            setDefaultCloseOperation(EXIT_ON_CLOSE);//设置当关闭窗体时，退出系统
            jbInit();                              //对窗体内组件进行初始化设置的方法
        } catch (Exception exception) {
            exception.printStackTrace();
        }
    }
    // 初始化窗体内的组件，抛出 java.lang.Exception 异常
    private void jbInit() throws Exception {
        //初始化窗体的代码
        ……
    }
    // 【重置】按钮动作事件处理方法
    public void resetButton_actionPerformed(ActionEvent actionEvent) {
        //处理【重置】按钮动作事件的代码
        ……
    }
    // “登录”按钮动作事件处理方法
    public void loginButton_actionPerformed(ActionEvent actionEvent) {
        //处理【登录】按钮动作事件的代码
        ……
    }
    // 将窗体居中显示
    public void centerWindow(JFrame frame){
        //将窗体居中显示的代码
        ……
    }
    // 内部类，作为【退出系统】按钮的动作事件监听器
    class LoginFrame_exitButton_actionAdapter implements ActionListener {
        //事件监听器的实现代码
    }
    // 内部类，作为【登录】按钮的动作事件监听器
    class LoginFrame_jButton1_actionAdapter implements ActionListener {
        //事件监听器的实现代码
    }
    // 内部类，作为【重置】按钮的动作事件监听器
    class LoginFrame_resetButton_actionAdapter implements ActionListener {
        //事件监听器的实现代码
    }
}
```

可以清楚地看出，在登录窗体类 LoginFrame 类中，完成了对登录界面的布局初始化，并响应相应的登录操作、重置操作和退出系统操作。其中对登录界面的布局初始化的代码如下所示。

```
// 初始化窗体内的组件，抛出 java.lang.Exception 异常
private void jbInit() throws Exception {
```



```
        // 调用自定义的 createImageIcon 方法，方法参数为图像路径，创建界面显示的小图标
        image1 = createImageIcon("images/butffler.jpg");
        image2 = createImageIcon("images/username.gif");
        image3 = createImageIcon("images/password.gif");
        image4 = createImageIcon("images/warning.gif");
        image5 = createImageIcon("images/role2.jpg");

        // 设置标签上的文本和图像
        ……

        //设置登录按钮、重置按钮和退出系统按钮
        ……
        loginButton.addActionListener(new  LoginFrame_jButton1_actionAdapter
(this));                                                          //注册监听器
        ……
        resetButton.addActionListener(new  LoginFrame_resetButton_actionAdapter
(this));                                                          //注册监听器
        ……
        exitButton.addActionListener(new LoginFrame_exitButton_actionAdapter
(this));                                                          //注册监听器

        // 设置角色选择组合框
        roleName.setOpaque(false);
        roleModel.addElement("--请选择您登录的角色--");
        roleModel.addElement("班 主 任");
        roleModel.addElement("教务专员");
        roleModel.addElement("人事专员");
        roleName.setModel(roleModel);

        // 设置内容面板的首选大小，并添加组件到内容面板
        ……
    }
```

在 jbInit()方法中，完成了界面显示组件的创建和布局，并为相应的“登录”按钮、“重置”按钮和“退出”按钮注册了相应的监听器，初始设置了角色选择组合框中的选择项。其中调用自定义的方法 createImageIcon()方法用来读取图像文件，以显示在界面标签上。createImageIcon()方法的定义如下。

```
    // 返回一个 ImageIcon 对象,如果文件路径无效，则返回 null.
    public ImageIcon createImageIcon(String path) {
        BufferedImage img = null;
        try {
            img = ImageIO.read(new File(path));              //读取图像文件
        } catch (IOException ex) {
            System.err.println("找不到文件: " + path);
        }
        if(img != null){
            return new ImageIcon(img);
        }else{
            return null;
        }
    }
```

当用户进行登录操作时，如果需要重置输入文本框中的内容，则单击【重置】按钮，调用下面的事件响应代码进行重置处理：

```
// 【重置】按钮动作事件处理方法
public void resetButton_actionPerformed(ActionEvent actionEvent) {
    warnLabel.setText("提示：");
    infoLabel.setText("请输入账号和密码登录！");
    userName.setText("");
    passWord.setText("");
    roleName.setSelectedIndex(0);
}
```

当用户输入账号和密码后，单击【登录】按钮进行登录操作，这时会调用如下的事件响应代码进行登录操作，包括阻止空账户和空密码、验证账户和密码的正确性、根据登录用户的角色决定调用打开的功能模块等。其实现代码如下。

```
// 【登录】按钮动作事件处理方法
public void loginButton_actionPerformed(ActionEvent actionEvent) {
    String roleid="";
    int roleselected=roleName.getSelectedIndex();
    if(roleselected==0){    //如果没有选择角色
        warnLabel.setForeground(Color.RED);
        warnLabel.setText("警告：");
        infoLabel.setForeground(Color.RED);
        infoLabel.setText("您没有选择合适的角色，请选择后登录！");
        roleName.requestFocus();//角色选择框获得输入焦点
        return;
    }
    System.out.println("roleselected="+roleselected);
    switch(roleselected){
        case 1:
            roleid="%D%";
            break;        //班主任登录,账号:301a004
        case 2:
            roleid="%E%";
            break;        //教务专员登录,账号:301a008
        case 3:
            roleid="%C%";
            break;        //人事专员登录，账号:301a009
    }
    //第一步：创建一个 bean，封装登录信息：
    //用户名（userid:301A007),密码(userpwd:111111),用户角色(roleid:A)
    String userPwd=new String(this.passWord.getPassword());
    UserInfoBean user=new UserInfoBean();   //这是一个封装登录用户信息的类
    user.setUserId(this.userName.getText()); //将用户输入的信息封装到类对象中
    user.setUserPwd(userPwd);
    user.setRoleId(roleid);
    LoginAction loginAction=new LoginAction(); //这是用来对用户账号、密码和角色进行验证的类
    user=loginAction.verify(user);          //验证参数 user 代表的登录用户信息
        if(user!=null){                     //如果用户验证通过
            if(userPwd.equals("111111")){    //如果是初始密码登录，要提示更改密码
                JOptionPane.showMessageDialog(this, "为了安全，请及时更改原始密
```

```
码！", "警告",
                                                JOptionPane.WARNING_MESSAGE);
            }
            infoLabel.setText("OK,您已通过验证！");
            //根据用户的角色，打开不同的管理界面
            switch(roleselected){
               case 1:                                    //如果是班主任登录
                  openClassManagerWindow(user);      //调用此自定义方法，打开
班级管理模块
                  break;
               case 2:                           //如果是教务专员登录
                  openEduAdminWindow(user); //调用此自定义方法，打开教务管理模块
                  break;
               case 3:                           //如果是人事专员登录
                  openHRWindow(user);        //调用此自定义方法，打开人事管理模块
                  break;
             }
         }
         else{                                    //如果用户不合法
            warnLabel.setForeground(Color.RED);  //将警告信息标签的字体颜色设
为红色
            warnLabel.setText("警告：");       //改变“提示”标签上文本为“警告:”
            infoLabel.setForeground(Color.RED);  //将信息提示标签的字体颜色设
为红色
            infoLabel.setText("您的账号或密码或角色选择不正确！");
            userName.requestFocus();              //用户账号文本框获得输入焦点
            userName.setSelectionColor(new Color(200,200,230));
            userName.selectAll();                        //设置文本框选中
            passWord.setText("");                        //清空密码框
         }
    }//end loginButton_actionPerformed()
```

从上面的代码可以看出，当用户的账户和密码验证通过以后，系统会根据登录用户的角色，打开相应的管理界面，达到控制操作权限的目的。下面是打开相应管理模块的方法定义：

```
    // 自定义方法，打开班主任登录窗口
    public void openClassManagerWindow(UserInfoBean user){
       // 创建班主任管理窗体对象
       ClassManagerFrame classManagerFrame = new ClassManagerFrame(user);
       classManagerFrame.pack();
       // 窗体居中
       centerWindow(classManagerFrame);
    }
    // 自定义方法，打开教务专员登录窗口
    public void openEduAdminWindow(UserInfoBean user){
       EduAdminFrame eduAdminFrame = new EduAdminFrame(user);
       eduAdminFrame.pack();
       // 窗体居中显示
       centerWindow(eduAdminFrame);
    }
    //自定义方法，打开人事专员登录窗口
    public void openHRWindow(UserInfoBean user){
```

```
    HRFrame hrFrame = new HRFrame(user);
    hrFrame.pack();
    // 窗体居中显示
    centerWindow(hrFrame);
}
```

通常用户希望看到启动的程序出现在屏幕的中央，因此需要将窗体居中显示。将窗体居中显示的代码如下所示。

```
// 将窗体居中显示
public void centerWindow(JFrame frame){
   Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
   Dimension frameSize = frame.getSize();
   if (frameSize.height > screenSize.height) {
       frameSize.height = screenSize.height;
   }
   if (frameSize.width > screenSize.width) {
       frameSize.width = screenSize.width;
   }
   frame.setLocation((screenSize.width - frameSize.width) / 2,
                  (screenSize.height - frameSize.height) / 2-50);
   frame.setVisible(true);                    //显示窗体
   this.setVisible(false);                    //当前登录窗体隐藏
   this.invalidate();
}
```

在 jbInit()方法中，分别为 loginButton、resetButton 和 exitButton 这三个按钮注册了监听器。在这里，监听器是通过定义内部类实现的。三个监听器的定义如下。

```
// 内部类，作为【退出系统】按钮的动作事件监听器
class LoginFrame_exitButton_actionAdapter implements ActionListener {
    private LoginFrame adaptee;
    LoginFrame_exitButton_actionAdapter(LoginFrame adaptee) {
        this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent e) {
        adaptee.exitButton_actionPerformed(e);
    }
}
// 内部类，作为【登录】按钮的动作事件监听器
class LoginFrame_jButton1_actionAdapter implements ActionListener {
    private LoginFrame adaptee;
    LoginFrame_jButton1_actionAdapter(LoginFrame adaptee) {
        this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent actionEvent) {
        adaptee.loginButton_actionPerformed(actionEvent);
    }
}
// 内部类，作为【重置】按钮的动作事件监听器
class LoginFrame_resetButton_actionAdapter implements ActionListener {
    private LoginFrame adaptee;
    LoginFrame_resetButton_actionAdapter(LoginFrame adaptee) {
        this.adaptee = adaptee;
    }
```



```
    public void actionPerformed(ActionEvent actionEvent) {
        adaptee.resetButton_actionPerformed(actionEvent);
    }
}
}
```

在上面的代码中，还用到了如下几个类。

- UserInfoBean 类：这是一个用户信息类，用来封装登录用户的信息（账号、密码和角色信息）。
- LoginAction 类：这是一个登录业务类，用来连接数据库，验证登录用户信息正确与否。
- ClassManagerFrame 类：这是一个班主任管理窗体界面类，是 JFrame 的子类。
- EduAdminFrame 类：这是一个教务专员管理窗体界面类，是 JFrame 的子类。
- HRFrame 类：这是一个人事专员管理窗体界面类，是 JFrame 的子类。

后面的三个窗体类，会在后面的相应章节进行详细介绍。下面重点介绍登录模块相关的 UserInfoBean 类和 LoginAction 类。

代表用户登录信息的类 UserInfoBean 类的定义如下所示。

```
package accpedu;
public class UserInfoBean {
    private int userNo;                          //用户序号
    private String userId;                       //用户账号（userid:301A007)
    private String userPwd;                      //密码(userpwd:111111)
    private String ifFirst;                      //是否第一次登录(isfirst:是)
    private String roleId;                       //用户角色(roleid:accpno_01)
    public UserInfoBean() {
    }
    public int getUserNo() {                     //获得用户序号
        return userNo;
    }
    public void setUserNo(int userNo) {          //设置用户序号
        this.userNo = userNo;
    }
    public String getUserId() {                  //获得用户账号
        return userId;
    }
    public void setUserId(String userId) {  //设置用户账号
        this.userId = userId;
    }
    public String getUserPwd() {                 //获得用户密码
        return userPwd;
    }
    public void setUserPwd(String userPwd) {     //设置用户密码
        this.userPwd = userPwd;
    }
    public String getRoleId() {                      //获得用户角色
        return roleId;
    }
    public void setRoleId(String roleId) {       //设置用户角色
        this.roleId = roleId;
```

```
        }
    }
```

LoginAction 类是一个业务类，用来处理登录相关业务，如连接数据库，账号、密码和角色验证。其定义如下所示。

```
    package accpedu;
    import java.sql.Connection;
    import java.sql.DriverManager;
    import java.sql.PreparedStatement;
    import java.sql.ResultSet;
    import java.sql.SQLException;
    public class LoginAction {
        Connection con=null;
        //构造函数，用来加载驱动程序，并和数据库创建连接 S
        public LoginAction() {
            try{
                System.out.println("正在加载驱动程序......");
                Class.forName("com.mysql.jdbc.Driver");
                System.out.println("加载驱动程序成功");
            }catch(ClassNotFoundException e){
                System.out.println("加载驱动程序失败");
            }//end try-catch
        }//end 构造函数
        // 和数据库建立连接。返回连接对象
        public Connection getConnection(){
            Connection con=null;
            String          url="jdbc:mysql://localhost:3306/accp?useUnicode=
true&characterEncoding=GBK";
            String user="dbuser";                          //与 mysql 的设置相同，代
表 mysql 客户端用户
            String password="1234";                        //mysql 客户端登录密码
            try{
                System.out.println("正在和数据库建立连接......");
                con=DriverManager.getConnection(url, user, password);
                System.out.println("数据库连接成功");
            }catch(SQLException e){
                System.out.println("无法和数据库建立连接");
            }//end try-catch
            return con;
    }
        //在数据库中验证用户合法性
        public UserInfoBean verify(UserInfoBean user){
            PreparedStatement ps=null;
            ResultSet rs=null;
            if(con==null){
                con=this.getConnection();
            }
            UserInfoBean newUser=null;
            String sqlstr="select * from userinfo where userid=? and userpwd=?
and userrole like ?";
            try{
                ps=con.prepareStatement(sqlstr);
                ps.setString(1, user.getUserId());   //设置参数 1：用户账号 id
```



```
                ps.setString(2, user.getUserPwd());  //设置参数 2：用户密码
                ps.setString(3,user.getRoleId());    //设置参数 3：角色 id
                rs=ps.executeQuery();                //执行查询
                if(rs.next()){                       //如果用户提供的账号、密码和角
色正确
                    newUser=user;
                    newUser.setUserNo(rs.getInt(1)); //设置账户序号
                }
            }catch (SQLException ex) {
                System.out.println("验证用户合法性时出错");
            }finally{                                //关闭数据库连接对象
                try{
                    if(rs!=null)    rs.close();
                    if(ps!=null)    ps.close();
                    if(con!=null)   con.close();
                }catch(SQLException ex){
                    //do nothing
                }
            }
            return newUser;                  //返回完整的登录用户信息
        }//end method:verify()
    }
```

在 LoginFrame 类的“登录”按钮动作事件处理方法中，有如下两行代码：

```
if(userPwd.equals("111111")){            //如果是初始密码登录，要提示更改密码
    JOptionPane.showMessageDialog(this, "为了安全，请及时更改原始密码！", "警告",
                                          JOptionPane.WARNING_MESSAGE);
}
```

即当用户以统一分配的初始密码“111111”登录时，要提示用户为了安全，要及时更改密码，如图 27.25 所示。

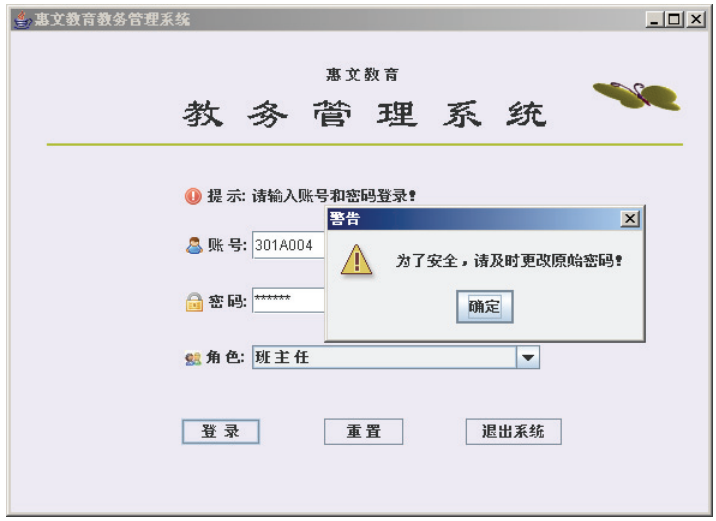


图 27.25　提示用户更改原始密码

下一节讲解如何设计菜单及在菜单添加修改用户密码的菜单项。

27.5.3 实现菜单栏

作为事件驱动的系统，菜单所起的作用是非常明显的。一个好的菜单，能起到锦上添花的作用。本节就讲述教务管理系统的菜单设计及其代码实现。

教务管理系统的各个管理子模块各不相同，但菜单都是统一的，用来实现系统管理、用户密码修改和提供作者及版本信息等，如图 27.26 所示。



图 27.26 菜单栏及菜单

在教务管理系统中，单独创建一个 MenuBar 类，并实现一个 createMenuBar()方法，该方法返回一个功能完备的菜单栏。这样在任何一个管理模块窗口，都可以创建 MenuBar 对象并调用其 createMenuBar()方法，获得已经实现了的菜单栏。MenuBar 类的实现如下所示。

```
package accpedu;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class MenuBar {
    JFrame frame = null;                              //声明一个 JFrame 对象
    UserInfoBean userinfo = null;                     //声明一个 UserInfoBean 对象
    JMenuBar jMenuBar1 = new JMenuBar();              //定义一个菜单栏对象
    JMenu jMenuFile = new JMenu();                    //定义“系统管理”的菜单
    JMenuItem jMenuFileExit = new JMenuItem();
                                      //定义“系统管理”菜单的“退出系统”菜单项
    JMenu jMenuHelp = new JMenu();  //定义“帮助”的菜单
    JMenuItem jMenuHelpAbout = new JMenuItem();
                                      //定义“帮助”菜单的“关于”菜单项
    JMenu jMenuPwd = new JMenu();   //定义“密码维护”菜单
    JMenuItem jMenuResetPwd = new JMenuItem();
                                      //定义“密码维护”菜单的“更改密码”菜单项
    public MenuBar(JFrame frame,UserInfoBean userinfo) { //构造器，接收两个参数
        this.frame = frame;                           //接收窗体对象
        this.userinfo = userinfo;                     //接收用户对象
        //设置【系统管理】菜单
        jMenuFile.setText("系统管理");
        jMenuFileExit.setText("退出系统");
        jMenuFileExit.addActionListener(new       //添加事件监听器
                                  Frame_jMenuFileExit_ActionAdapter());
        jMenuFile.add(jMenuFileExit);             //添加“退出系统”菜单项
        //设置【帮助】菜单
        jMenuHelp.setText("帮助");
        jMenuHelpAbout.setText("关于");
        jMenuHelpAbout.addActionListener(new      //添加事件监听器
                                   Frame_jMenuHelpAbout_ActionAdapter
(frame));
        jMenuHelp.add(jMenuHelpAbout);            //添加“关于”菜单项
```



```
            //设置【密码维护】菜单
            jMenuPwd.setText("密码维护");
            jMenuResetPwd.setText("更改密码");
            jMenuResetPwd.addActionListener(new      //添加事件监听器
                                        Frame_ jMenuResetPwd_actionAdapter
(frame,userinfo));
            jMenuPwd.add(jMenuResetPwd);             //添加“更改”菜单项
            //将菜单添加到菜单栏
            jMenuBar1.setBackground(new Color(237, 234, 255));
            jMenuBar1.setBorder(BorderFactory.createEtchedBorder());
            jMenuBar1.add(jMenuFile);
            jMenuBar1.add(jMenuPwd);
            jMenuBar1.add(jMenuHelp);
        }
        public JMenuBar createMenuBar(){            //返回菜单栏
            return this.jMenuBar1;
        }
    }
    // 【退出系统】菜单项的事件监听器
    class Frame_jMenuFileExit_ActionAdapter implements ActionListener {
            public void actionPerformed(ActionEvent actionEvent) {
            System.exit(0);                          //响应菜单项事件，退出系统
        }
    }
    // 【更改密码】菜单项的事件监听器
    class Frame_ jMenuResetPwd_actionAdapter implements ActionListener {
        private JFrame adaptee;
        private UserInfoBean userinfo;
        Frame_ jMenuResetPwd_actionAdapter(JFrame adaptee,UserInfoBean userinfo) {
            this.adaptee = adaptee;
            this.userinfo = userinfo;
        }
        //单击【更改密码】菜单
        public void actionPerformed(ActionEvent e) {
            // 创建自定义的密码修改对话框对象，并修改密码
            UpdatePwdDialog dlg = new UpdatePwdDialog(adaptee,"修改密码",true,
userinfo);
            // 设置密码修改框居中显示
            Dimension dlgSize = dlg.getPreferredSize();
            Dimension frmSize = adaptee.getSize();
            Point loc = adaptee.getLocation();
            dlg.setLocation((frmSize.width - dlgSize.width) / 2 + loc.x,
                        (frmSize.height - dlgSize.height) / 2 + loc.y);
            dlg.setModal(true);
            dlg.pack();
            dlg.setVisible(true);
        }
    }
    // 【关于】菜单项的事件监听器
    class Frame_jMenuHelpAbout_ActionAdapter implements ActionListener {
        JFrame adaptee;
        Frame_jMenuHelpAbout_ActionAdapter(JFrame adaptee) {
            this.adaptee = adaptee;
```

```
        }
        // 当单击【关于】菜单项时
        public void actionPerformed(ActionEvent actionEvent) {
            // 创建自定义的关于作者信息的对话框对象
            AboutBox dlg = new AboutBox(adaptee);
            // 设置【关于】对话框居中显示
            Dimension dlgSize = dlg.getPreferredSize();
            Dimension frmSize = adaptee.getSize();
            Point loc = adaptee.getLocation();
            dlg.setLocation((frmSize.width - dlgSize.width) / 2 + loc.x,
                        (frmSize.height - dlgSize.height) / 2 + loc.y);
            dlg.setModal(true);
            dlg.pack();
            dlg.setVisible(true);
        }
    }
```

在 MenuBar 类中，使用了两个自定义的类，一个是用来更改密码的对话框类 UpdatePwdDialog，一个是用来提供关于作者信息的 AboutBox 对话框类。接下来介绍 UpdatePwdDialog 类及 AboutBox 类的设计和实现。

27.5.4 实现修改密码模块

修改密码的界面比较简单，使用对话框类 JDialog 即可。在教务管理系统中，通过派生自 JDialog 类，生成自定义的对话框类 UpdatePwdDialog。其界面如图 27.27 所示。

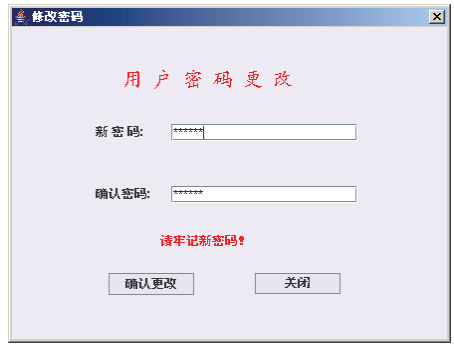


图 27.27 用户密码更改界面

其代码实现如下。

```
    package accpedu;
    import java.awt.*;
    import javax.swing.*;
    import java.awt.event.*;
    public class UpdatePwdDialog extends
JDialog {
        JPanel panel1 = new JPanel();
        JLabel jLabel1 = new JLabel();                    //声明“密码”标签
        JLabel jLabel2 = new JLabel();                    //声明“确认密码”标签
        JPasswordField jPasswordField1 = new JPasswordField();//声明“密码框”
        JPasswordField jPasswordField2 = new JPasswordField();//声明“确认密码框”
        JLabel jLabel3 = new JLabel();
        JLabel jLabel4 = new JLabel();
        JButton jButton1 = new JButton();                 //声明“确认更改”按钮
        JButton jButton2 = new JButton();                 //声明“关闭”按钮
        UserInfoBean userinfo=null;                       //声明封装用户信息的类
        GridBagLayout gridBagLayout1 = new GridBagLayout();//声明网格包布局对象
        public UpdatePwdDialog(Frame owner, String title, boolean modal,
UserInfoBean userinfo) {
            super(owner, title, modal);
            this.userinfo=userinfo;                       //获得登录的用户信息
            try {
```

```
            setDefaultCloseOperation(DISPOSE_ON_CLOSE);//设置当关闭窗体时释放资源
            jbInit();
            pack();
        } catch (Exception exception) {
            exception.printStackTrace();
        }
    }
    public UpdatePwdDialog() {
        this(null, "密码更改对话框", true,null);
    }
    private void jbInit() throws Exception {
    // 对界面进行初始化的代码
    ……
}
    // 单击【确认按钮】，更改密码
public void jButton1_actionPerformed(ActionEvent e) {
    // 响应事件代码
    ……
}
//单击【关闭】按钮，关闭修改密码的对话框
    public void jButton2_actionPerformed(ActionEvent e) {
        //响应事件代码
        ……
    }
    // 响应【确认更改】按钮动作事件的监听器类
class UpdatePwdDialog_jButton1_actionAdapter implements ActionListener {
    ……
        public void actionPerformed(ActionEvent e) {
            //进行事件处理的代码
            ……
        }
    }
    // 响应【关闭】按钮动作事件的监听器类
    class UpdatePwdDialog_jButton1_actionAdapter implements ActionListener {
    ……
        public void actionPerformed(ActionEvent e) {
            //进行事件处理的代码
            ……
        }
    }
```

其中 jbInit()方法用来对完成创建界面组件、进行界面布局、注册相应的事件监听器。其实现代码如下所示。

```
private void jbInit() throws Exception {
    panel1.setLayout(gridBagLayout1);
    panel1.setBackground(new Color(237, 234, 255));
    jLabel1.setText("新 密 码:");
    jLabel2.setText("确认密码:");
    jLabel3.setFont(new java.awt.Font("楷体_GB2312", Font.PLAIN, 20));
    jLabel3.setForeground(Color.red);
    jLabel3.setText("用 户 密 码 更 改");
    jLabel4.setForeground(Color.red);
    jLabel4.setText("请牢记新密码! ");
```

```
        jPasswordField1.setText("111111");
        jPasswordField2.setText("111111");
        jButton1.setBackground(new Color(230, 230, 255));
        jButton1.setText("确认更改");
        //为确认按钮添加事件监听器
        jButton1.addActionListener(new    UpdatePwdDialog_jButton1_actionAdapter
(this));
        jButton2.setBackground(new Color(230, 230, 255));
        jButton2.setText("关闭");
        //为关闭按钮添加事件监听器
        jButton2.addActionListener(new    UpdatePwdDialog_jButton2_actionAdapter
(this));
        getContentPane().add(panel1);
        panel1.add(jLabel3, new GridBagConstraints(0, 0, 3, 1, 0.0, 0.0
                , GridBagConstraints.WEST, GridBagConstraints.NONE,
                new Insets(36, 112, 0, 96), 22, 10));
        panel1.add(jLabel1, new GridBagConstraints(0, 1, 1, 1, 0.0, 0.0
                , GridBagConstraints.WEST, GridBagConstraints.NONE,
                new Insets(25, 84, 0, 0), 11, 3));
        panel1.add(jPasswordField1, new GridBagConstraints(1, 1, 2, 1, 1.0, 0.0
                , GridBagConstraints.WEST, GridBagConstraints.HORIZONTAL,
                new Insets(25, 11, 0, 90), 110, -5));
        panel1.add(jLabel2, new GridBagConstraints(0, 2, 1, 1, 0.0, 0.0
                , GridBagConstraints.WEST, GridBagConstraints.NONE,
                new Insets(39, 84, 0, 0), 10, 6));
        panel1.add(jLabel4, new GridBagConstraints(1, 3, 2, 1, 0.0, 0.0
                , GridBagConstraints.WEST, GridBagConstraints.NONE,
                new Insets(24, 0, 0, 156), 7, 4));
        panel1.add(jPasswordField2, new GridBagConstraints(1, 2, 2, 1, 1.0, 0.0
                , GridBagConstraints.WEST, GridBagConstraints.HORIZONTAL,
                new Insets(40, 11, 0, 90), 110, -5));
        panel1.add(jButton2, new GridBagConstraints(2, 4, 1, 1, 0.0, 0.0
                , GridBagConstraints.CENTER, GridBagConstraints.NONE,
                new Insets(21, 45, 44, 90), 26, -7));
        panel1.add(jButton1, new GridBagConstraints(0, 4, 2, 1, 0.0, 0.0
                , GridBagConstraints.CENTER, GridBagConstraints.NONE,
                new Insets(21, 97, 44, 0), 0, -6));
    }
```

其中响应按钮单击事件进行业务处理的代码如下所示。

```
    // 单击【确认】按钮，更改密码
    public void jButton1_actionPerformed(ActionEvent e) {
        String pwd1=new String(jPasswordField1.getPassword());
                                            //获得第一个密码框中输入的新密码
        String pwd2=new String(jPasswordField2.getPassword());
                                            //获得第二个密码框中的确认密码
        if(!pwd1.equals(pwd2)||pwd1.length()==0){//判断，如果两次输入密码不一致
            jLabel4.setText("您两次输入的密码不一致或为空，请重新输入!");
            jPasswordField1.setText("111111");
            jPasswordField2.setText("111111");
            return;                         //返回
        }
        UserAction userAction=new UserAction(); //调用自定义用户信息修改业务类
```



```
            int result=userAction.updateUserPwd(userinfo.getUserId(),pwd1);
                                                    //调用其修改数据库中保存的密码方法
            if(result!=0){
                JOptionPane.showMessageDialog(this, "密码修改成功！", "消息",
                                        JOptionPane.INFORMATION_MESSAGE);
            }
            else{
                JOptionPane.showMessageDialog(this, "密码修改失败！", "警告",
                                        JOptionPane.WARNING_MESSAGE);
                return;
            }
            jPasswordField1.setText("");
            jPasswordField2.setText("");
            dispose();
        }
        //单击【关闭】按钮，关闭修改密码的对话框
        public void jButton2_actionPerformed(ActionEvent e) {
            dispose();
        }
    }
```

【确认】按钮所注册的动作事件监听器和【确认更改】按钮所注册的事件监听器都是以内部类的形式定义的，其实现如下所示。

```
// 响应【关闭】按钮动作事件的监听器类
class UpdatePwdDialog_jButton2_actionAdapter implements ActionListener {
    private UpdatePwdDialog adaptee;
    UpdatePwdDialog_jButton2_actionAdapter(UpdatePwdDialog adaptee) {
        this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent e) {
        adaptee.jButton2_actionPerformed(e);
    }
}
// 响应【确认更改】按钮动作事件的监听器类
class UpdatePwdDialog_jButton1_actionAdapter implements ActionListener {
    private UpdatePwdDialog adaptee;
    UpdatePwdDialog_jButton1_actionAdapter(UpdatePwdDialog adaptee) {
        this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent e) {
        adaptee.jButton1_actionPerformed(e);
    }
}
```

在更改用户密码时，调用了一个自定义的业务类 UserAction 类。这是符合 MVC 模式的业务与表现相分离的程序设计模式。业务类 UserAction 类的实现如下所示。

```
package accpedu;
import java.sql.PreparedStatement;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.util.ArrayList;
public class UserAction {
```

```
    public UserAction() {
    }
    //在数据库中增加某一用户信息
    public int addUser(String userid){
        //进行数据库连接并添加用户信息的代码
        ……
    }
    //在数据库中删除某一用户信息
    public int delUser(String userid){
        //进行数据库连接并删除用户信息的代码
        ……
    }
    //在数据库中查找某一用户信息
    public UserBean getUser(String userid){
        //进行数据库连接并查找某一用户信息的代码
        ……
    }
    //在数据库中修改某一用户权限
    public int updateUserRole(String userid,String role){
        //进行数据库连接并修改某一用户权限的代码
        ……
    }
    //在数据库中修改某一用户密码
    public int updateUserPwd(String userid,String userpwd){
        //进行数据库连接并修改用户密码的代码
        ……
    }
}
```

其中向数据库中增加某一用户信息的代码定义在一个单独的方法 addUser(String userid)中。该方法接收一个用户的账号，根据账号向用户账号信息表中添加一个账号，初始密码默认为“111111”。该方法的定义如下所示。

```
    //在数据库中增加某一用户信息
    public int addUser(String userid){
        String sqlstr = "insert into userinfo(userid,userpwd,userrole) values
(?,?)";
        int result = 0;
        DBConnection dbcon = new DBConnection();
        Connection con = dbcon.getConnection();              // 和数据库建立连接
        PreparedStatement ps = null;
        try {
            ps = con.prepareStatement(sqlstr);
            ps.setString(1, userid);
            ps.setString(2, "111111");
            ps.setString(3, "");
            result = ps.executeUpdate();
        } catch (SQLException ex) {
            System.out.println("增加用户信息时出错");
        } finally {                                          //关闭数据库连接对象
            try {
                if (ps != null)
                    ps.close();
```



```
            if (con != null)
                con.close();
            } catch (SQLException ex) {
                //do nothing
            }
        }
        return result;
    }
```

其中从数据库中删除某一用户信息的代码定义在一个单独的方法 delUser(String userid) 中。该方法接收一个用户的账号，根据账号从用户账号信息表中删除一个账户信息。该方法的定义如下所示。

```
    //在数据库中删除某一用户信息
    public int delUser(String userid){
        String sqlstr = "delete from userinfo  where rtrim(userid)=?";
        int result = 0;
        DBConnection dbcon = new DBConnection();          //创建数据库连接对象
        Connection con = dbcon.getConnection();           // 和数据库建立连接
        PreparedStatement ps = null;
        try {
            ps = con.prepareStatement(sqlstr);            //创建预编译语句对象
            ps.setString(1, userid);
            result = ps.executeUpdate();
        } catch (SQLException ex) {
            System.out.println("删除用户信息时出错");
        } finally {                                       //关闭数据库连接对象
            try {
                if (ps != null)
                    ps.close();
                if (con != null)
                    con.close();
            } catch (SQLException ex) {
                //do nothing
            }
        }
        return result;
    }
```

其中在数据库中查找某一用户信息的代码定义在一个单独的方法 getUser(String userid) 中。该方法接收一个用户的账号，根据账号在用户账号信息表中查找指定的账号。该方法的定义如下所示。

```
    //在数据库中查找某一用户信息
    public UserBean getUser(String userid){
        String sqlstr = "select * from userinfo where rtrim(userid)=?";
        UserBean userInfo=new UserBean();
        DBConnection dbcon = new DBConnection();    //创建数据库连接对象
        Connection con = dbcon.getConnection();     // 和数据库建立连接
        PreparedStatement ps = null;
        ResultSet rs=null;
        try {
            ps = con.prepareStatement(sqlstr);      //创建预编译语句对象
            ps.setString(1, userid);                //设置预编译语句中第一个参数
```

```
            rs = ps.executeQuery();                    //执行查询，并返回查询结果集
            if(rs.next()){                             //如果查询有结果
                userInfo.setUserNo(rs.getInt(1));
                                    //从结果集中获取查询结果，封装到用户信息对象中
                userInfo.setUserId(rs.getString(2));
                userInfo.setUserPwd(rs.getString(3));
                userInfo.setUserRole(rs.getString(4));
            }
        } catch (SQLException ex) {
            System.out.println("查找用户信息时出错");
        } finally {                                    //关闭数据库连接对象
            try {
                if(rs!=null)
                    rs.close();
                if (ps != null)
                    ps.close();
                if (con != null)
                    con.close();
            } catch (SQLException ex) { }
        }
        return userInfo;
    }
```

其中在数据库中修改某一用户权限的代码定义在一个单独的方法 updateUserRole (String userid,String role)中。该方法接收一个用户的账号和新的角色，根据账号在用户账号信息表中修改角色。该方法的定义如下所示。

```
    //在数据库中修改某一用户权限
    public int updateUserRole(String userid,String role){
        String sqlstr = "update userinfo set userrole=? where rtrim (userid)=?";
        int result = 0;
        DBConnection dbcon = new DBConnection();
        Connection con = dbcon.getConnection();          // 和数据库建立连接
        PreparedStatement ps = null;
        try {
            ps = con.prepareStatement(sqlstr);
            ps.setString(1,role);
            ps.setString(2, userid);
            result = ps.executeUpdate();
        } catch (SQLException ex) {
            System.out.println("更新用户权限时出错");
        } finally {                                      //关闭数据库连接对象
            try {
                if (ps != null)
                    ps.close();
                if (con != null)
                    con.close();
            } catch (SQLException ex) { }
        }
        return result;
    }
```

其中在数据库中修改某一用户的密码定义在一个单独的方法 updateUserPwd (String userid,String userpwd)中。该方法接收一个用户的账号和新的密码，根据账号在用户账号信



息表中修改密码。该方法的定义如下所示。

```
    //在数据库中修改某一用户密码
    public int updateUserPwd(String userid,String userpwd){
        String sqlstr = "update userinfo set userpwd=? where rtrim(userid)=?";
        int result = 0;
        DBConnection dbcon = new DBConnection();
        Connection con = dbcon.getConnection();      // 和数据库建立连接
        PreparedStatement ps = null;
        try {
            ps = con.prepareStatement(sqlstr);
            ps.setString(1,userpwd);
            ps.setString(2, userid);
            result = ps.executeUpdate();
        } catch (SQLException ex) {
            System.out.println("更改用户密码时出错");
        } finally {                                          //关闭数据库连接对象
            try {
                if (ps != null)
                    ps.close();
                if (con != null)
                    con.close();
            } catch (SQLException ex) {
                //do nothing
            }
        }
        return result;
    }
}
```

凡是与用户信息有关的业务操作，都封装在 UserAction 类中，包括向数据库中增加用户、从数据库中删除用户、修改用户信息等。在这里，只使用了其中的 updateUserPwd 方法来修改用户的密码。其他方法后续的操作中会用到。

27.5.5　实现“关于”对话框

“关于”对话框主要用来提示一些帮助信息，如软件使用说明、作者信息等。在本系统中，“关于”对话框是 JDialog 类的子类，主要用来提示有关作者的基本信息和软件版本信息等，如图 27.28 所示。



图 27.28　【关于】对话框

其实现代码如下所示。

```
    package accpedu;
    import java.awt.*;
    import java.awt.event.*;
    import javax.swing.*;
    import
com.borland.jbcl.layout.XYLayout;
    import com.borland.jbcl.layout.*;
    public class AboutBox extends JDialog implements ActionListener {
        JPanel panel1 = new JPanel();
```

```
        JPanel panel2 = new JPanel();
        JPanel insetsPanel1 = new JPanel();
        JPanel insetsPanel2 = new JPanel();
        JPanel insetsPanel3 = new JPanel();
        JButton button1 = new JButton();
        JLabel imageLabel = new JLabel();
        JLabel label1 = new JLabel();
        JLabel label2 = new JLabel();
        JLabel label3 = new JLabel();
        ImageIcon image1 = new ImageIcon();
        BorderLayout borderLayout2 = new BorderLayout();
        FlowLayout flowLayout1 = new FlowLayout();
        GridLayout gridLayout1 = new GridLayout();
        String product = "惠文教育教务管理系统";
        String version = "1.0";
        String copyright = "Copyright (c) 2008";
        String comments = "";
        JLabel jLabel1 = new JLabel();
        XYLayout xYLayout1 = new XYLayout();
        public AboutBox(Frame parent) {
            super(parent);
            try {
                setDefaultCloseOperation(DISPOSE_ON_CLOSE);
                jbInit();
            } catch (Exception exception) {
                exception.printStackTrace();
            }
        }
        public AboutBox() {
            this(null);
        }
        //  初始化组件，声明抛出 java.lang.Exception
        private void jbInit() throws Exception {
            image1 = new ImageIcon(accpedu.LoginFrame.class.getResource("about.
png"));
            imageLabel.setIcon(image1);
            setTitle("关于");
            panel1.setLayout(xYLayout1);
            panel2.setLayout(borderLayout2);
            insetsPanel1.setLayout(flowLayout1);
            insetsPanel2.setLayout(flowLayout1);
            insetsPanel2.setBackground(new Color(237, 234, 255));
            insetsPanel2.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
            gridLayout1.setRows(4);
            gridLayout1.setColumns(1);
            label1.setText(product);
            label2.setText(version);
            label3.setText(copyright);
            insetsPanel3.setLayout(gridLayout1);
            insetsPanel3.setBackground(new Color(237, 234, 255));
            insetsPanel3.setBorder(BorderFactory.createEmptyBorder(10, 60, 10, 10));
            button1.setBackground(new Color(230, 230, 255));
            button1.setText("OK");
            button1.addActionListener(this);
```



```
        insetsPanel1.setBackground(new Color(237, 234, 255));
        panel1.setBackground(new Color(237, 234, 255));
        jLabel1.setFont(new java.awt.Font("隶书", Font.PLAIN, 20));
        jLabel1.setForeground(new Color(219, 0, 0));
        jLabel1.setText("作者：辛立伟    问题反馈:13253584578");
        panel2.setBackground(new Color(237, 234, 255));
        insetsPanel2.add(imageLabel, null);
        panel1.add(label1, new XYConstraints(120, 74, -1, -1));
        panel2.add(insetsPanel2, BorderLayout.WEST);
        getContentPane().add(panel1, null);
        insetsPanel3.add(label2, null);
        insetsPanel3.add(label3, null);
        panel1.add(insetsPanel1, new XYConstraints(0, 219, 400, 81));
        insetsPanel1.add(button1, null);
        panel1.add(jLabel1, new XYConstraints(12, 188, 372, 31));
        panel1.add(panel2, new XYConstraints(0, 0, 400, 75));
        panel1.add(insetsPanel3, new XYConstraints(64, 93, -1, 95));
        setResizable(true);
    }
    // 处理“OK”按钮事件，关于对话框，并释放资源
    public void actionPerformed(ActionEvent actionEvent) {
        if (actionEvent.getSource() == button1) {
            dispose();
        }
    }
}
```

27.5.6　实现人事管理模块

本节介绍人事专员管理窗体界面类 HRFrame 类的设计和实现，它是 JFrame 的子类。其界面设计如图 27.29 所示。



图 27.29　人事管理模块初始界面

功能设计如下。

- 初始登录界面，以表格形式显示全体员工的基本信息，如图 27.29 所示。
- 在表格中单击选中任一员工，在下面的组件中显示相对应的详细的员工信息，如图 27.30 所示。



图 27.30　显示详细员工信息

- 在【部门】组合框中选择任一部门，并单击【查看】按钮，会在表格中自动列出所选择部门的员工信息，如图 27.31 所示。



图 27.31　显示部门员工信息



- 单击【入职新员工】按钮，则所有文本框清空，【入职新员工】按钮上的文本变为“取消”，【员工离职】按钮和【信息修改】按钮变为禁用状态，【保存】按钮变为可用状态。在详细文本框中输入新入职员工的信息，单击【保存】按钮，会自动将新员工信息保存到数据库中，如图 27.32 所示。

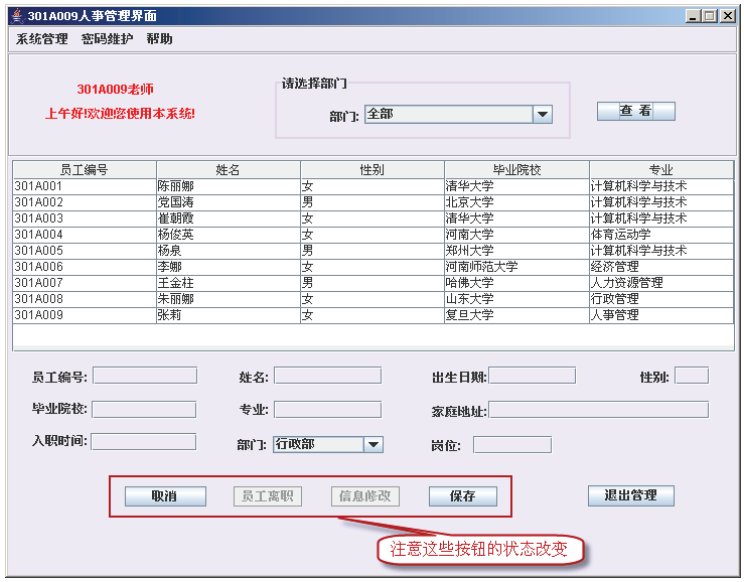


图 27.32　显示部门员工信息

- 先在表格中选中要操作的教员，单击【员工离职】按钮，会从数据库及表格中删除要离职的员工；或单击【信息修改】按钮，显示教员详细信息的文本框会变为可编辑，修改信息以后，单击【保存】按钮，会保存修改后的员工信息。
- 单击【退出管理】按钮，会关闭当前窗体，重新回到登录界面。

其代码实现如下所示。

```
package accpedu;
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
import com.borland.jbcl.layout.XYLayout;
import com.borland.jbcl.layout.*;
import javax.swing.border.TitledBorder;
import java.awt.event.*;
import java.util.*;
import javax.swing.event.*;
public class HRFrame extends JFrame implements ListSelectionListener{
    BorderLayout borderLayout1 = new BorderLayout();
    JPanel jPanel1 = new JPanel();
    XYLayout xYLayout1 = new XYLayout();
    TitledBorder titledBorder1 = new TitledBorder("");
    JPanel jPanel2 = new JPanel();
    JPanel tablePanel = new JPanel();
    XYLayout xYLayout2 = new XYLayout();
    JScrollPane jScrollPane1 = new JScrollPane();   //表格要放在滚动面板中
```

```
    JTable teacherTable = new JTable();              //声明表对象

//创建组件的代码，此处省略
……

    ArrayList teacherlist = null;        //动态数组对象，存储所有员工信息
    ArrayList departmentlist = null;     //动态数组对象，存储所有部门信息]
    UserInfoBean userinfo = null;        //登录用户信息对象
    ClassInfoBean classInfo = null;      //班级信息对象
    ListSelectionModel selectionMode = null;
                                         //声明JTable的ListSelectionModel模型
    String department = "hw04";          //部门，默认为学术部
    javax.swing.JComboBox jComboBox1 = new JComboBox();
    public HRFrame(UserInfoBean user) {
        this.userinfo = user;            //将登录用户的信息传递给教务管理窗口
        try {
            jbInit();
        } catch (Exception exception) {
            exception.printStackTrace();
        }
        sayHello();                      //设置欢迎词
        this.departmentlist = initDepartComboBox(); //初始化 departComboBox
        initDetailDepart();                         //初始化部门信息
        this.teacherlist = initTeacherTable();      //初始化 Table 表
    }
    //创建组件并进行初始化组件、注册事件监听器
    private void jbInit() throws Exception {
        ...
        addteacherButton.addActionListener(new //为【新入职员工】按钮注册事件监听器
                                HRFrame_addteacherButton_actionAdapter
(this));
        ...
        delteacherButton.addActionListener(new
                                           //为【离职员工】按钮注册事件监听器
                                HRFrame_delteacherButton_actionAdapter
(this));
        ...
        modifyteacherButton.addActionListener(new
                                           //为【修改信息】按钮注册事件监听器
              HRFrame_modifyteacherButton_actionAdapter(this));
        ...
        saveteacherButton.addActionListener(new //为【保存】按钮注册事件监听器
              HRFrame_saveteacherButton_actionAdapter(this));
        ...
        exitmanageButton.addActionListener(new //为【退出系统】按钮注册事件监听器
              HRFrame_exitmanageButton_actionAdapter(this));
        ...
        teacherInfoButton.addActionListener(new //为【查看】按钮注册事件监听器
              HRFrame_teacherInfoButton_actionAdapter(this));
        ...
        departComboBox.addActionListener(new     //为【部门】组合框注册事件监听器
                                  HRFrame_departComboBox_actionAdapter
(this));
```



```
    ...
}
//设置欢迎词
public void sayHello(){
    ...
}
//在部门下拉列表中初始化部门信息
 public ArrayList initDepartComboBox(){
     ...
}
//初始化详细信息(detail)部分的部门信息
public void initDetailDepart(){
     ...
}
//初始化 Table 表
public ArrayList initTeacherTable(){
     ArrayList list=this.getTeacherInfo();
     ...
     selectionMode.addListSelectionListener(this);
     return list;
}
//表格选取响应事件
/*当用户选取表格数据时会触发 ListSelectionEvent,
* 我们实现 ListSelectionListener 界面来处理这一事件.ListSelectionListener 界
* 面只定义一个方法,那就是 valueChanged().
*/
public void valueChanged(ListSelectionEvent e) {
     ...
}
//当单击【退出管理】按钮时，响应的操作
 public void exitmanageButton_actionPerformed(ActionEvent e) {
     ...
}
//单击【查看】按钮，刷新员工信息表格的内容
public void teacherInfoButton_actionPerformed(ActionEvent e) {
     ...
}
//增加一个新的员工信息
public void addteacherButton_actionPerformed(ActionEvent e) {
     ...
}

//当单击【员工离职】按钮时，响应删除员工信息操作
public void delteacherButton_actionPerformed(ActionEvent e) {
     ...
}

//当单击【修改信息】按钮时，响应修改员工信息的操作
public void modifyteacherButton_actionPerformed(ActionEvent e) {
     ...
}
//保存一个新的班级信息
public void saveteacherButton_actionPerformed(ActionEvent e) {
```

```
        ...
    }
}

// 定义事件监听器类
...
```

其中 jbInit()方法用来创建组件并进行初始化组件、注册事件监听器，其实现代码如下所示。

```
private void jbInit() throws Exception {
    titledBorder1 = new TitledBorder(javax.swing.BorderFactory.
                        createLineBorder(new Color(227, 212,234), 2), "
请选择部门");
    getContentPane().setLayout(borderLayout1);
    jPanel1.setLayout(xYLayout1);
    jLabel1.setForeground(Color.red);
    jLabel1.setText("XXX 老师");
    jLabel2.setForeground(Color.red);
    jLabel2.setText("上午好!欢迎您使用本系统!");
    jPanel2.setBackground(new Color(237, 234, 255));
    jPanel2.setBorder(titledBorder1);
    jPanel2.setLayout(xYLayout3);
    tablePanel.setBackground(new Color(237, 234, 255));
    tablePanel.setMinimumSize(new Dimension(725, 415));
    tablePanel.setPreferredSize(new Dimension(725, 415));
    tablePanel.setLayout(xYLayout2);
    teacherTable.setMaximumSize(new Dimension(500, 300));
    teacherTable.setPreferredSize(new Dimension(100, 165));
    jLabel5.setText("员工编号:");
    jLabel6.setText("姓名:");
    jLabel7.setText("出生日期:");
    jLabel8.setText("性别:");
    jLabel9.setText("毕业院校:");
    jLabel10.setText("专业:");
    jLabel11.setText("家庭地址:");
    jLabel12.setText("入职时间:");
    jLabel19.setText("部门:");
    jLabel25.setText("岗位:");
    addteacherButton.setAlignmentY((float) 0.0);
    addteacherButton.setMargin(new Insets(2, 2, 2, 2));
    addteacherButton.setText("入职新员工");
    addteacherButton.addActionListener(new  //为【新入职员工】按钮注册事件监听器
                        HRFrame_addteacherButton_actionAdapter(this));
    delteacherButton.setMargin(new Insets(2, 2, 2, 2));
    delteacherButton.setText("员工离职");
    delteacherButton.addActionListener(new  //为【离职员工】按钮注册事件监听器
                        HRFrame_delteacherButton_actionAdapter(this));
    modifyteacherButton.setMargin(new Insets(2, 2, 2, 2));
    modifyteacherButton.setText("信息修改");
    modifyteacherButton.addActionListener(new//为【修改信息】按钮注册事件监听器
           HRFrame_modifyteacherButton_actionAdapter(this));
    saveteacherButton.setEnabled(false);
```



```
saveteacherButton.setText("保存");
saveteacherButton.addActionListener(new //为【保存】按钮注册事件监听器
       HRFrame_saveteacherButton_actionAdapter(this));
exitmanageButton.setText("退出管理");
exitmanageButton.addActionListener(new //为【退出系统】按钮注册事件监听器
       HRFrame_exitmanageButton_actionAdapter(this));
teacherInfoButton.setMargin(new Insets(2, 2, 2, 2));
teacherInfoButton.setText("查　看");
teacherInfoButton.addActionListener(new //为【查看】按钮注册事件监听器
       HRFrame_teacherInfoButton_actionAdapter(this));
teacher_5.setOpaque(false);
teacher_5.setEditable(false);
teacher_6.setOpaque(false);
teacher_6.setEditable(false);
teacher_7.setOpaque(false);
teacher_7.setEditable(false);
teacher_4.setOpaque(false);
teacher_4.setEditable(false);
teacher_10.setOpaque(false);
teacher_10.setEditable(false);
jLabel3.setText("    ");
teacher_1.setOpaque(false);
teacher_1.setEditable(false);
this.setResizable(false);
teacher_2.setOpaque(false);
teacher_2.setEditable(false);
teacher_3.setOpaque(false);
teacher_3.setEditable(false);
teacher_8.setOpaque(false);
teacher_8.setEditable(false);
jLabel13.setText("部门:");
jLabel14.setText("jLabel14");
jComboBox1.setEnabled(false);
jComboBox1.setOpaque(false);
departComboBox.addActionListener(new        //为部门组合框注册事件监听器
                       HRFrame_departComboBox_actionAdapter (this));
jPanel1.setBackground(new Color(237, 234, 255));
jPanel1.setBorder(BorderFactory.createEtchedBorder());
this.getContentPane().add(jPanel1, java.awt.BorderLayout.NORTH);
jPanel1.add(jLabel2, new XYConstraints(35, 45, 149, 27));
jPanel1.add(jLabel1, new XYConstraints(67, 23, 94, 23));
this.getContentPane().add(tablePanel, java.awt.BorderLayout.CENTER);
jScrollPane1.getViewport().add(teacherTable);
jPanel1.add(jLabel3, new XYConstraints(345, 81, -1, -1));
tablePanel.add(jLabel5, new XYConstraints(24, 210, 59, -1));
tablePanel.add(jLabel7, new XYConstraints(421, 210, 57, -1));
tablePanel.add(teacher_3, new XYConstraints(477, 208, 88, 18));
tablePanel.add(jLabel8, new XYConstraints(628, 210, 35, -1));
jPanel1.add(teacherInfoButton, new XYConstraints(583, 46, 78, 19));
jPanel1.add(jPanel2, new XYConstraints(261, 18, 298, 67));
jPanel2.add(departComboBox, new XYConstraints(85, 9, 187, 19));
jPanel2.add(jLabel13, new XYConstraints(51, 12, -1, -1));
MenuBar menuBar = new MenuBar(this,this.userinfo);
jMenuBar1 = menuBar.createMenuBar();            //创建菜单栏
```

```
        setJMenuBar(jMenuBar1);                                    //设置菜单栏
        tablePanel.add(jScrollPane1, new XYConstraints(4, 3, 719, 191));
        tablePanel.add(jLabel6, new XYConstraints(230, 210, 35, -1));
        tablePanel.add(teacher_4, new XYConstraints(662, 208, 35, 18));
        tablePanel.add(jLabel12, new XYConstraints(24, 274, 59, 17));
        tablePanel.add(jLabel19, new XYConstraints(228, 277, -1, 18));
        tablePanel.add(teacher_10, new XYConstraints(462, 277, 79, 18));
        tablePanel.add(jLabel25, new XYConstraints(421, 278, -1, -1));
        tablePanel.add(modifyteacherButton, new XYConstraints(321, 327, 68, 20));
        tablePanel.add(delteacherButton, new XYConstraints(224, 327, 68, 20));
        tablePanel.add(saveteacherButton, new XYConstraints(418, 327, 68, 20));
        tablePanel.add(addteacherButton, new XYConstraints(116, 327, 81, 20));
        tablePanel.add(jLabel9, new XYConstraints(24, 242, 59, 17));
        tablePanel.add(teacher_5, new XYConstraints(83, 242, 105, 18));
        tablePanel.add(jLabel10, new XYConstraints(230, 243, 34, 17));
        tablePanel.add(teacher_6, new XYConstraints(264, 242, 108, 18));
        tablePanel.add(jLabel11, new XYConstraints(421, 244, 57, -1));
        tablePanel.add(teacher_1, new XYConstraints(84, 208, 105, 18));
        tablePanel.add(teacher_7, new XYConstraints(477, 242, 220, 18));
        tablePanel.add(teacher_8, new XYConstraints(82, 274, 106, 18));
        tablePanel.add(teacher_2, new XYConstraints(264, 208, 108, 18));
        tablePanel.add(jComboBox1, new XYConstraints(263, 277, 110, 17));
        tablePanel.add(exitmanageButton, new XYConstraints(576, 326, 86, 21));
    }
```

当用户登录以后，会首先出现一个欢迎信息，并根据用户登录的时间，给予相应的问候语。如果是中午十二点之前登录，会提示“上午好”；如果是中午十二点以后登录，会提示“下午好”。其实现代码如下所示。

```
    //设置欢迎词
    public void sayHello(){
        Calendar calendar = new GregorianCalendar();    //获取日期管理对象
        Date trialTime = new Date();    //获取当前日期
        calendar.setTime(trialTime);
        this.setTitle(userinfo.getUserId()+"人事管理界面");
                                        //设置标题栏，在标题栏中显示登录用户信息
        jLabel1.setText(userinfo.getUserId()+"老师");    //根据登录用户设置欢迎词
        if(calendar.get(Calendar.HOUR_OF_DAY)>12){
                                        //根据登录时间显示“上午好”或“下午好”
            jLabel2.setText("下午好!欢迎您使用本系统!");
        }
    }
```

当人事专员刚登录人事管理模块时，在部门下拉列表中应该初始化显示所有的部门信息，在详细信息部分，也应该出现详细的部门信息，在表格中初始显示所有的员工信息。其实现代码如下所示。

```
    //在部门下拉列表中初始化部门信息
     public ArrayList initDepartComboBox(){
          DefaultComboBoxModel    defaultComboBoxModel=new    DefaultComboBox
Model();
          defaultComboBoxModel.addElement("全部");
          DepartmentAction departmentAction=new DepartmentAction();
                                                        //访问部门信息的业务类
```



```
        ArrayList list=departmentAction.findAllDepartments();
                                                    //获取所有的部门
        Iterator it=list.iterator();                //获得迭代器
        while(it.hasNext()){                        //迭代所有部门
            DepartmentInfoBean departmentinfo=(DepartmentInfoBean)it.next();
            defaultComboBoxModel.addElement(departmentinfo.getDepartment
Name());
        }
        departComboBox.setModel(defaultComboBoxModel);
        departComboBox.setSelectedIndex(0);
        return list;
    }
    //初始化详细信息(detail)部分的部门信息
    public void initDetailDepart(){
        DefaultComboBoxModel   defaultComboBoxModel=new   DefaultComboBox
Model();
        DepartmentAction departmentAction=new DepartmentAction();
                                                    //创建部门业务类对象
        ArrayList list=departmentAction.findAllDepartments();
                                                    //调用相关业务方法
        Iterator it=list.iterator();
        while(it.hasNext()){
            DepartmentInfoBean departmentinfo=(DepartmentInfoBean)it.next();
            defaultComboBoxModel.addElement(departmentinfo.getDepartment
Name());
        }
        jComboBox1.setModel(defaultComboBoxModel);
        jComboBox1.setSelectedIndex(0);
    }
    //初始化 Table 表
    public ArrayList initTeacherTable(){
        ArrayList list=this.getTeacherInfo();
        DefaultTableModel defaultTableModel1=new DefaultTableModel();
        defaultTableModel1.setColumnCount(0);
        defaultTableModel1.setRowCount(0);
        //添加表头
        defaultTableModel1.addColumn("员工编号");
        defaultTableModel1.addColumn("姓名");
        defaultTableModel1.addColumn("性别");
        defaultTableModel1.addColumn("毕业院校");
        defaultTableModel1.addColumn("专业");
        //添加表格内容
        TeacherInfoBean teacherinfo=null;
        int i=0;
        if(list!=null){
            Iterator it=list.iterator();
            while(it.hasNext()){
                defaultTableModel1.setNumRows(i+1);
                teacherinfo=(TeacherInfoBean)it.next();
               defaultTableModel1.setValueAt(teacherinfo.getTeacherId(),
i,0);
                defaultTableModel1.setValueAt(teacherinfo.getTeacher
Name(),i,1);
```

```
                    defaultTableModel1.setValueAt(teacherinfo.getTeacher
Sex(),i,2);
                    defaultTableModel1.setValueAt(teacherinfo.getTeacher
School(),i,3);
                    defaultTableModel1.setValueAt(teacherinfo.getTeacher
Tech(),i,4);
                    i++;
                }
          }else{
                System.out.println("error:HRFrame.initTeacherTable()");
          }
          teacherTable.setModel((TableModel)defaultTableModel1);
          //使得表格的选取是以 cell 为单位,而不是以列为单位.若你没有写此行,则在选取表格数
          //据时以整列为单位
          teacherTable.setCellSelectionEnabled(true);
          //取得 table 的 ListSelectionModel.
          selectionMode= teacherTable.getSelectionModel();
          selectionMode.setSelectionMode(ListSelectionModel.SINGLE_
SELECTION);                                                    //设置为单选
          teacherTable.revalidate();
          selectionMode.addListSelectionListener(this);
          return list;
    }
```

当用户在员工信息表中选择一个员工时（选中其中一行），这时该员工的详细信息应出现在表格下方的各个文本框中。这是通过为表格添加一个表格选取响应事件监听器 ListSelectionListener 来实现的。该监听器会接收并处理用户选取表格数据时触发的 ListSelectionEvent 事件。其实现代码如下所示。

```
    //表格选取响应事件
    /*当用户选取表格数据时会触发 ListSelectionEvent,
    * 我们实现 ListSelectionListener 监听器来处理这一事件.ListSelectionListener 监听
器中
    * 只定义一个方法,那就是 valueChanged().
    */
    public void valueChanged(ListSelectionEvent e) {
          int row= teacherTable.getSelectedRow();      //获取用户选择的行
          if(row<0){
                row=0;
          }
          if(row>=0 && teacherlist.size()>0){
                TeacherInfoBean teacherinfo = (TeacherInfoBean) teacherlist.
get(row);                                                  //获取选择的用户信息
                //给文本框赋值
                teacher_1.setText(teacherinfo.getTeacherId());
                teacher_2.setText(teacherinfo.getTeacherName());
                teacher_3.setText(teacherinfo.getTeacherBirthday());
                teacher_4.setText(teacherinfo.getTeacherSex());
                teacher_5.setText(teacherinfo.getTeacherSchool());
                teacher_6.setText(teacherinfo.getTeacherTech());
                teacher_7.setText(teacherinfo.getTeacherAddress());
                teacher_8.setText(teacherinfo.getTeacherInDate());
                jComboBox1.setSelectedIndex(row);
```



```
            teacher_10.setText(teacherinfo.getTeacherDuty());
        }
}
```

当用户单击【退出管理】按钮时，会调用下面的方法进行处理：

```
//当单击【退出管理】按钮时，响应的操作
public void exitmanageButton_actionPerformed(ActionEvent e) {
    this.setVisible(false);
    this.invalidate();
    this.dispose();
    LoginFrame loginFrame=new LoginFrame();        //创建登录窗体
    loginFrame.pack();
    // 登录窗体居中显示
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    Dimension frameSize = loginFrame.getSize();
    if (frameSize.height > screenSize.height) {
        frameSize.height = screenSize.height;
    }
    if (frameSize.width > screenSize.width) {
        frameSize.width = screenSize.width;
    }
    loginFrame.setLocation((screenSize.width - frameSize.width) / 2,
                        (screenSize.height - frameSize.height) / 2);
    loginFrame.setVisible(true);
}
```

当用户单击【查看】按钮时，会调用下面的方法进行处理：

```
//单击【查看】钮，刷新员工信息表格的内容
public void teacherInfoButton_actionPerformed(ActionEvent e) {
    this.teacherlist=initTeacherTable();
    this.clearTeacherInfo();
    this.repaint();
}
//清空所有文本框
public void clearTeacherInfo(){
    teacher_1.setText("");
    teacher_2.setText("");
    teacher_3.setText("");
    teacher_4.setText("");
    teacher_5.setText("");
    teacher_6.setText("");
    teacher_7.setText("");
    teacher_8.setText("");
    jComboBox1.setSelectedIndex(0);
    teacher_10.setText("");
}
```

当用户单击【入职新员工】按钮时，会调用下面的方法进行处理：

```
//增加一个新的员工信息
public void addteacherButton_actionPerformed(ActionEvent e) {
    if(e.getActionCommand().equals("入职新员工")){
        addteacherButton.setText("取消");
        setAllTextFieldEditable(true);          //将文本框设置为可编辑
```

```
            clearTeacherInfo();                          //首先清空所有文本框
            delteacherButton.setEnabled(false);     //【删除】按钮失效
            modifyteacherButton.setEnabled(false); //【修改】按钮失效
            saveteacherButton.setEnabled(true);
        }else{
            addteacherButton.setText("入职新员工");
            setAllTextFieldEditable(false);          //将文本框设置为不可编辑
            delteacherButton.setEnabled(true);      //【删除】按钮生效
            modifyteacherButton.setEnabled(true);  //【修改】按钮生效
            saveteacherButton.setEnabled(false);    //【保存】按钮失效
        }
    }
    //设置文本框状态
    public void setAllTextFieldEditable(boolean isEnable){
        teacher_1.setEditable(isEnable);
        teacher_2.setEditable(isEnable);
        teacher_3.setEditable(isEnable);
        teacher_4.setEditable(isEnable);
        teacher_5.setEditable(isEnable);
        teacher_6.setEditable(isEnable);
        teacher_7.setEditable(isEnable);
        teacher_8.setEditable(isEnable);
        jComboBox1.setEnabled(isEnable);
        teacher_10.setEditable(isEnable);
    }
```

当用户单击【员工离职】按钮时，会调用下面的方法进行处理：

```
    //当单击【员工离职】按钮时，响应删除员工信息操作
    public void delteacherButton_actionPerformed(ActionEvent e) {
        String teacherId= teacher_1.getText().trim();
        if(teacherId.equals("")){
            JOptionPane.showMessageDialog(this,"请先选择要删除的员工！",
                                "警告",JOptionPane.WARNING_MESSAGE);
            teacher_1.requestFocus();                    //重新获得输入焦点
            return;
        }
        TeacherAction teacherAction=new TeacherAction();
                                                         //员工信息操作的业务类
        int line=0;
        int result = JOptionPane.showConfirmDialog(this,"您真的要删除此员工信息
吗？",
                                                  "删除确认", JOptionPane.YES_NO_
OPTION);
        if (result == JOptionPane.YES_OPTION){      //选择：Yes or OK
            line=teacherAction.delTeacher(teacherId);  //删除员工
            if(line==0)
                JOptionPane.showMessageDialog(this,"删除失败，请检查！",
                                                  "  警  告  ",JOptionPane.WARNING_
MESSAGE);
            else
                JOptionPane.showMessageDialog(this,"删除成功！",
                                                  "  成  功  ",JOptionPane.WARNING_
```

```
MESSAGE);
            //重新初始化 Table 表
            this.teacherlist=initTeacherTable();
            this.repaint();
        }//end if
    }
```

根据人事专员所选择的部门，从数据库中查找该部门所有员工的信息，并保存到一个动态数组中。这是通过自定义的方法 getTeacherInfo()方法来实现的，其代码如下所示。

```
    //从员工信息表中获得 ArrayList 对象
    public ArrayList getTeacherInfo(){
        int selectedDepalrtment=departComboBox.getSelectedIndex();
                                                    //获得用户选择的部门
        String department="%";
        switch(selectedDepalrtment){
            case 0:department="%";break;
            case 1:department="hw01";break;
            case 2:department="hw02";break;
            case 3:department="hw03";break;
            case 4:department="hw04";break;
            case 5:department="hw05";break;
            case 6:department="hw06";break;
            case 7:department="hw07";break;
            case 8:department="hw08";break;
            default:department="%";
        }
        ArrayList list=new ArrayList();
        TeacherAction teacherAction = new TeacherAction();
        list = teacherAction.findTeachersByDepart(department);
                                                    //查找指定部门的所有员工
        return list;
    }
```

当用户单击【修改信息】按钮时，会调用下面的方法进行处理：

```
    //当单击【修改信息】按钮时，响应修改员工信息的操作
    public void modifyteacherButton_actionPerformed(ActionEvent e) {
        String teacherId= teacher_1.getText().trim();
        if(teacherId.equals("")){
            JOptionPane.showMessageDialog(this,"请先选择要修改的员工！",
                                          "  警  告  ",JOptionPane.WARNING_
MESSAGE);
            teacher_1.requestFocus();
            return;
        }
        //当进行信息修改时
        if(e.getActionCommand().equals("信息修改")){
            modifyteacherButton.setText("取消");
            setAllTextFieldEditable(true);          //将文本框设置为可编辑
            delteacherButton.setEnabled(false);     //【删除】按钮失效
            addteacherButton.setEnabled(false);     //【增加】按钮失效
            saveteacherButton.setEnabled(true);
        }else{                                      //如果是“取消“信息修改时
            modifyteacherButton.setText("信息修改");
```

```
            setAllTextFieldEditable(false);          //将文本框设置为不可编辑
            delteacherButton.setEnabled(true);       //【删除】按钮生效
            addteacherButton.setEnabled(true);       //【增加】按钮生效
            saveteacherButton.setEnabled(false);
        }
    }
```

当用户单击【保存员工信息】按钮时，会调用下面的方法进行处理：

```
    //保存一个新的员工信息
    public void saveteacherButton_actionPerformed(ActionEvent e) {
        String teacherId= teacher_1.getText().trim();
        if(teacherId.equals("")){
            JOptionPane.showMessageDialog(this,"员工编号不能为空！",
                                          "  警  告  ",JOptionPane.WARNING_
MESSAGE);
            teacher_1.requestFocus();
            return;
        }
        //如果是保存新增加的员工信息,将员工信息同时增加到员工信息表
        if(addteacherButton.isEnabled()){
            TeacherInfoBean teacherinfo=createTeacher();
                                          //将员工信息封装到 bean 当中
            TeacherAction teacherAction=new TeacherAction();
            int result = teacherAction.addTeacher(teacherinfo);
                                          //将指定员工信息添加到数据库中
            if(result==0){
                JOptionPane.showMessageDialog(this,"添加员工信息失败，请检查输入是
否正确！",
                                          "警告",JOptionPane.WARNING_
MESSAGE);
                return;
            }else{
                JOptionPane.showMessageDialog(this,"添加员工信息成功",
                                          " 消 息 ",JOptionPane.INFORMATION_
MESSAGE);
                setAllTextFieldEditable(false);  //重新将文本框设置为不可编辑
                saveteacherButton.setEnabled(false);     //【保存】按钮失效
                addteacherButton.setText("增加新班");
                delteacherButton.setEnabled(true);       //【删除】按钮生效
                modifyteacherButton.setEnabled(true);    //【修改】按钮生效
            }
        }
        else if(modifyteacherButton.isEnabled()){ //如果是保存修改过的班级信息
            TeacherInfoBean teacherinfo=createTeacher();
                                                  //将员工信息封装到 bean 当中
            TeacherAction teacherAction=new TeacherAction();
            int result1=new TeacherAction().delTeacher(teacherId);
                                                  //删除指定员工
            if(result1==0){
                JOptionPane.showMessageDialog(this,"无法修改，请检查输入是否正确！",
                                          "警告",JOptionPane.WARNING_
MESSAGE);
```



```
                return;
            }
            int result2=new TeacherAction().addTeacher(teacherinfo);
            if(result2==0){
                JOptionPane.showMessageDialog(this,"修改员工信息失败，请检查输入是
否正确！",
                                              "警告",JOptionPane.WARNING_
MESSAGE);
                return;
            }else{
                JOptionPane.showMessageDialog(this,"修改员工信息成功",
                                              "消 息 ",JOptionPane.INFORMATION_
MESSAGE);
                setAllTextFieldEditable(false);   //重新将文本框设置为不可编辑
                saveteacherButton.setEnabled(false);      //【保存】按钮失效
                delteacherButton.setEnabled(true);        //【删除】按钮生效
                addteacherButton.setEnabled(true);        //【增加】按钮生效
                modifyteacherButton.setText("修改");
            }
        }
        //重新初始化 Table 表
        this.teacherlist=this.initTeacherTable();
        teacherTable.revalidate();
        tablePanel.repaint();
    }
    //将文本框信息封装到一个 TeacherInfoBean 当中
    public TeacherInfoBean createTeacher(){
        TeacherInfoBean teacherinfo=new TeacherInfoBean();
        //获取文本框的值
        teacherinfo.setTeacherId(teacher_1.getText().trim());
        teacherinfo.setTeacherName(teacher_2.getText().trim());
        teacherinfo.setTeacherBirthday(teacher_3.getText().trim());
        teacherinfo.setTeacherSex(teacher_4.getText().trim());
        teacherinfo.setTeacherSchool(teacher_5.getText().trim());
        teacherinfo.setTeacherTech(teacher_6.getText().trim());
        teacherinfo.setTeacherAddress(teacher_7.getText().trim());
        teacherinfo.setTeacherInDate(teacher_8.getText().trim());
        teacherinfo.setTeacherDepartment((String)jComboBox1.
getSelectedItem());
        teacherinfo.setTeacherDuty(teacher_10.getText().trim());
        return teacherinfo;
    }
}
```

与前面的代码功能模块相似，在这个模块中，程序也要大量地响应人事专员的各种操作，这也是通过为相应的组件添加事件监听器实现的。这些事件监听器都是单独定义的监听器类，通过构造监听器时传递进来的窗体对象进行事件处理方法回调，来响应用户的操作。这些监听器类的实现如下所示。

```
// 【信息修改】按钮的动作事件监听器
class HRFrame_modifyteacherButton_actionAdapter implements ActionListener {
    private HRFrame adaptee;
    HRFrame_modifyteacherButton_actionAdapter(HRFrame adaptee) {
```

```
        this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent e) {
        adaptee.modifyteacherButton_actionPerformed(e);
    }
}
// 【保存】按钮的动作事件监听器
class HRFrame_saveteacherButton_actionAdapter implements ActionListener {
    private HRFrame adaptee;
    HRFrame_saveteacherButton_actionAdapter(HRFrame adaptee) {
        this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent e) {
        adaptee.saveteacherButton_actionPerformed(e);
    }
}
// 【员工离职】按钮的动作事件监听器
class HRFrame_delteacherButton_actionAdapter implements ActionListener {
    private HRFrame adaptee;
    HRFrame_delteacherButton_actionAdapter(HRFrame adaptee) {
        this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent e) {
        adaptee.delteacherButton_actionPerformed(e);
    }
}
// 【新入职员工】按钮的动作事件监听器
class HRFrame_addteacherButton_actionAdapter implements ActionListener {
    private HRFrame adaptee;
    HRFrame_addteacherButton_actionAdapter(HRFrame adaptee) {
        this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent e) {
        adaptee.addteacherButton_actionPerformed(e);
    }
}
// 【查看】按钮的动作事件监听器
class HRFrame_teacherInfoButton_actionAdapter implements ActionListener {
    private HRFrame adaptee;
    HRFrame_teacherInfoButton_actionAdapter(HRFrame adaptee) {
        this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent e) {
        adaptee.teacherInfoButton_actionPerformed(e);
    }
}
// 【退出系统】按钮的动作事件监听器
class HRFrame_exitmanageButton_actionAdapter implements ActionListener {
    private HRFrame adaptee;
    HRFrame_exitmanageButton_actionAdapter(HRFrame adaptee) {
        this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent e) {
        adaptee.exitmanageButton_actionPerformed(e);
```



```
    }
}
```

在上面的类中，用到了 4 个类，分别如下。

- DepartmentInfoBean 类：这是一个部门信息类，用来封装部门的信息。
- TeacherInfoBean 类：这是一个员工信息类，用来封装员工的信息。
- DepartmentAction 类：这是一个对部门信息进行操作的业务类。用来连接数据库，进行与部门信息有关的操作。
- TeacherAction 类：这是一个对员工信息进行操作的业务类。用来连接数据库，进行与员工信息有关的操作。

在后面的相应的章节进行详细介绍。

27.5.7　实现 DepartmentInfoBean、TeacherInfoBean 信息封装类

通过创建部门信息封装类和员工信息封装类，可以很容易地生成其实例对象。在不同对象间通过传递信息封装类的实例对象，达到信息传递的目的。

其中，代表部门信息的 DepartmentInfoBean 类的实现如下。

```
package accpedu;
public class DepartmentInfoBean {
    private String departmentId;                        //声明代表部门编号的变量
    private String departmentName;                      //声明代表部门名称的变量
    public DepartmentInfoBean() {
    }
    public void setDepartmentId(String dId){            //设置部门编号
        this.departmentId=dId;
    }
    public String getDepartmentId(){                    //获取部门编号
        return this.departmentId;
    }
    public void setDepartmentName(String dName){        //设置部门名称
        this.departmentName=dName;
    }
    public String getDepartmentName(){                  //获取部门名称
        return this.departmentName;
    }
}
```

代表员工信息的 TeacherInfoBean 类的实现如下。

```
package accpedu;
public class TeacherInfoBean {
    private String teacherId;                           //员工编号
    private String teacherName;                         //员工姓名
    private String teacherBirthday;                     //员工出生日期
    private String teacherSex;                          //员工性别
    private String teacherSchool;                       //毕业院校
    private String teacherTech;                         //专业
    private String teacherAddress;                      //联系地址
    private String teacherInDate;                       //入职时间
    private String teacherDepartment;                   //所在部门
```

```
public TeacherInfoBean() {
}
//以下为对属性进行读写的 get/set 方法
public void setTeacherId(String tId){
    this.teacherId=tId;
}
public String getTeacherId(){
    return this.teacherId;
}
public void setTeacherName(String tName){
    this.teacherName=tName;
}
public String getTeacherName(){
    return this.teacherName;
}
public void setTeacherBirthday(String tBirthday){
    this.teacherBirthday=tBirthday;
}
public String getTeacherBirthday(){
    return this.teacherBirthday;
}
public void setTeacherSex(String tSex){
    this.teacherSex=tSex;
}
public String getTeacherSex(){
    return this.teacherSex;
}
public void setTeacherSchool(String tSchool){
    this.teacherSchool=tSchool;
}
public String getTeacherSchool(){
    return this.teacherSchool;
}
public void setTeacherTech(String tTech){
    this.teacherTech=tTech;
}
public String getTeacherTech(){
    return this.teacherTech;
}
public void setTeacherAddress(String tAddress){
    this.teacherAddress=tAddress;
}
public String getTeacherAddress(){
    return this.teacherAddress;
}
public void setTeacherInDate(String tInDate){
    this.teacherInDate=tInDate;
}
public String getTeacherInDate(){
    return this.teacherInDate;
}
public void setTeacherDepartment(String tDepartment){
    this.teacherDepartment=tDepartment;
}
```



```
    public String getTeacherDepartment(){
        return this.teacherDepartment;
    }
}
```

27.5.8　实现 DepartmentAction、TeacherAction 业务处理类

根据 MVC 设计模式，对后台数据进行操作的业务逻辑，用专门的业务处理类实现，可以达到解耦合、可扩展性强的目的。本系统就采用 MVC 的模式，将对部门信息的访问操作和对员工信息的访问操作，封装在专门的业务类 DepartmentAction 和 TeacherAction 中。

其中 DepartmentAction 类用来实现在数据库中查询所有部门信息的功能，代码实现如下所示。

```
package accpedu;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.sql.SQLException;
import java.sql.Connection;
public class DepartmentAction {
    public DepartmentAction() {
    }
    //在数据库中查询所有部门信息
    public ArrayList findAllDepartments(){
        ArrayList list=new ArrayList();
        String sqlstr="select * from departmentinfo";
        DBConnection dbcon = new DBConnection();
        Connection con = dbcon.getConnection();      // 和数据库建立连接
        PreparedStatement ps=null;
        ResultSet rs=null;
        try{
            ps=con.prepareStatement(sqlstr);
            rs=ps.executeQuery();     //执行查询操作，返回的结果保存在结果集中
            while(rs.next()){
                DepartmentInfoBean departmentinfo=new DepartmentInfoBean();
                //将每个班级信息保存到一个班级对象 departmentinfo 中
                departmentinfo.setDepartmentId(rs.getString(1));
                departmentinfo.setDepartmentName(rs.getString(2));
                //将查询出来的班级信息存放到一个动态数组当中
                list.add(departmentinfo);
            }
        }catch (SQLException ex) {
            System.out.println("查询部门时出错");
        }finally{                    //关闭数据库连接对象
            try{
                if(rs!=null)    rs.close();
                if(ps!=null)    ps.close();
                if(con!=null)   con.close();
            }catch(SQLException ex){
                //do nothing
            }
        }
```

```
        return list;                    //将集合了所查询到的部门信息动态数组返回
    }
}
```

在 findAllDepartments()方法中，将查询到的每一个部门都封装到部门信息类 DepartmentInfoBean 的实例对象中，然后将每一个对象（代表每一个部门）添加到一个动态数组中，最后返回此动态数组。在应用程序中，就可以在需要的地方调用此方法获得所有部门信息的数组列表。

TeacherAction 类用来实现在数据库中对员工信息进行处理的功能，包括：

- 在数据库中查询某部门已经是用户的员工信息。
- 在数据库中查询某部门非用户的员工信息。
- 在数据库中查询某部门员工信息。
- 向数据库中增加某一员工信息。
- 在数据库中删除某一员工信息。

上述每一个功能，都有相应的方法实现，其代码实现如下所示。

```
package accpedu;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Connection;
import java.util.ArrayList;
import java.sql.SQLException;
public class TeacherAction {
    public TeacherAction() {
}
    //在数据库中查询某部门已经是用户的员工信息
    public ArrayList findUserTeachersByDepart(String department){
        ArrayList list=new ArrayList();
        ...
        return list;                                    //返回数组列表
    }
    //在数据库中查询某部门非用户的员工信息
    public ArrayList findNonUserTeachersByDepart(String department){
        ArrayList list=new ArrayList();
        ...
        return list;
    }
    //在数据库中查询某部门员工信息
    public ArrayList findTeachersByDepart(String department){
        ArrayList list=new ArrayList();
        ...
        return list;
    }
    //向数据库中增加某一员工信息
    public int addTeacher(TeacherInfoBean teacherinfo){
        int result = 0;
        ...
        return result;
    }
    //在数据库中删除某一员工信息
    public int delTeacher(String teacherid){
```



```
        int result = 0;
        ...
        return result;
    }
}
```

其中在数据库中查询某部门已经是用户的员工信息的代码定义在一个单独的方法findUserTeachersByDepart(String department)中。该方法接收一个部门的编号，并查找该部门中所有已经是注册用户的员工信息。查询到的所有员工信息放入一个动态数组中，并返回。该方法的定义如下所示。

```
    //在数据库中查询某部门已经是用户的员工信息
    public ArrayList findUserTeachersByDepart(String department){
        ArrayList list=new ArrayList();
        //设置查询字符串
        String sqlstr="select * from teacherinfo where rtrim(teacherdepartment)
like ? "+
                      "and rtrim(teacherid) in ("+
                      "select userid from userinfo)";
        DBConnection dbcon = new DBConnection();          //创建连接对象
        Connection con = dbcon.getConnection();           //和数据库建立连接
        PreparedStatement ps=null;                        //声明预编译语句对象
        ResultSet rs=null;                                //声明结果集对象
        try{
            ps=con.prepareStatement(sqlstr);              //执行预编译命令
            ps.setString(1,department);                   //设置预编译命令参数
            rs=ps.executeQuery();                         //执行查询操作
            while(rs.next()){                             //如果查询结果不为空
                System.out.println("已经查找到此部门的所有已经是用户的员工信息");
                TeacherInfoBean teacherinfo=new TeacherInfoBean();
                                                  //创建封装员工信息的实例对象
                //将每个员工信息保存到一个教员对象 teacherinfo 中
                teacherinfo.setTeacherId(rs.getString(1));
                teacherinfo.setTeacherName(rs.getString(2));
                teacherinfo.setTeacherBirthday(rs.getString(3));
                teacherinfo.setTeacherSex(rs.getString(4));
                teacherinfo.setTeacherSchool(rs.getString(5));
                teacherinfo.setTeacherTech(rs.getString(6));
                teacherinfo.setTeacherAddress(rs.getString(7));
                teacherinfo.setTeacherInDate(rs.getString(8));
                teacherinfo.setTeacherDepartment(rs.getString(9));
                teacherinfo.setTeacherDuty(rs.getString(10));
                //将查询出来的班级信息存放到一个动态数组当中
                list.add(teacherinfo);
            }
        }catch (SQLException ex) {
            System.out.println("查询教员信息时出错 2");
        }finally{                                         //关闭数据库连接对象
            try{
                if(rs!=null)    rs.close();
                if(ps!=null)    ps.close();
                if(con!=null)   con.close();
            }catch(SQLException ex){
```

```
                //do nothing
            }
        }
        return list;                                    //返回数组列表
    }
```

其中在数据库中查询某部门非用户的员工信息的代码定义在一个单独的方法findNonUserTeachersByDepart(String department)中。该方法接收一个部门的编号，并查找该部门中所有还未注册成为用户的员工信息。查询到的所有员工信息放入一个动态数组中，并返回。该方法的定义如下所示。

```
    //在数据库中查询某部门非用户的员工信息
    public ArrayList findNonUserTeachersByDepart(String department){
        ArrayList list=new ArrayList();
        //设置查询字符串
        String sqlstr="select * from teacherinfo where rtrim(teacherdepartment)
like ? "+
                    "and rtrim(teacherid) not in ("+
                    "select userid from userinfo)";
        DBConnection dbcon = new DBConnection();        //创建连接对象
        Connection con = dbcon.getConnection();         //和数据库建立连接
        PreparedStatement ps=null;                      //声明预编译语句对象
        ResultSet rs=null;                              //声明结果集对象
        try{
            ps=con.prepareStatement(sqlstr);            //执行预编译命令
            ps.setString(1,department);                 //设置预编译命令参数
            rs=ps.executeQuery();                       //执行查询操作
            while(rs.next()){                           //如果查询结果不为空
                TeacherInfoBean teacherinfo=new TeacherInfoBean();
                //将每个员工信息保存到一个教员对象 teacherinfo 中
                teacherinfo.setTeacherId(rs.getString(1));
                teacherinfo.setTeacherName(rs.getString(2));
                teacherinfo.setTeacherBirthday(rs.getString(3));
                teacherinfo.setTeacherSex(rs.getString(4));
                teacherinfo.setTeacherSchool(rs.getString(5));
                teacherinfo.setTeacherTech(rs.getString(6));
                teacherinfo.setTeacherAddress(rs.getString(7));
                teacherinfo.setTeacherInDate(rs.getString(8));
                teacherinfo.setTeacherDepartment(rs.getString(9));
                teacherinfo.setTeacherDuty(rs.getString(10));
                //将查询出来的班级信息存放到一个动态数组当中
                list.add(teacherinfo);
            }
        }catch (SQLException ex) {
            System.out.println("查询教员信息时出错 2");
        }finally{                                       //关闭数据库连接对象
            try{
                if(rs!=null)    rs.close();
                if(ps!=null)    ps.close();
                if(con!=null)   con.close();
            }catch(SQLException ex){
                //do nothing
            }
```



```
        }
        return list;
    }
```

其中在数据库中查询指定某部门的员工信息的代码，定义在一个单独的方法findTeachersByDepart (String department)中。该方法接收一个部门的编号，并查找该部门中所有的员工信息。查询到的所有员工信息放入一个动态数组中，并返回。该方法的定义如下所示。

```
    //在数据库中查询某部门员工信息
    public ArrayList findTeachersByDepart(String department){
        ArrayList list=new ArrayList();
       String sqlstr="select * from teacherinfo where rtrim(teacherdepartment)
like ?";
        DBConnection dbcon = new DBConnection();
        Connection con = dbcon.getConnection();              // 和数据库建立连接
        PreparedStatement ps=null;
        ResultSet rs=null;
        try{
            ps=con.prepareStatement(sqlstr);
            ps.setString(1,department);
            rs=ps.executeQuery();
            while(rs.next()){
                TeacherInfoBean teacherinfo=new TeacherInfoBean();
                //将每个员工信息保存到一个教员对象 teacherinfo 中
                teacherinfo.setTeacherId(rs.getString(1));
                teacherinfo.setTeacherName(rs.getString(2));
                teacherinfo.setTeacherBirthday(rs.getString(3));
                teacherinfo.setTeacherSex(rs.getString(4));
                teacherinfo.setTeacherSchool(rs.getString(5));
                teacherinfo.setTeacherTech(rs.getString(6));
                teacherinfo.setTeacherAddress(rs.getString(7));
                teacherinfo.setTeacherInDate(rs.getString(8));
                teacherinfo.setTeacherDepartment(rs.getString(9));
                teacherinfo.setTeacherDuty(rs.getString(10));
                //将查询出来的班级信息存放到一个动态数组当中
                list.add(teacherinfo);
            }
        }catch (SQLException ex) {
            System.out.println("查询教员信息时出错");
        }finally{  //关闭数据库连接对象
            try{
                if(rs!=null)    rs.close();
                if(ps!=null)    ps.close();
                if(con!=null)   con.close();
            }catch(SQLException ex){
                //do nothing
            }
        }
        return list;
    }
```

其中向数据库中增加一个员工信息的代码，定义在一个单独的方法

addTeacher(TeacherInfoBean teacherinfo)中。该方法接收一个员工的信息，并将该员工的信息写入数据库中。该方法的定义如下所示。

```
//向数据库中增加某一员工信息
public int addTeacher(TeacherInfoBean teacherinfo){
    String sqlstr = "insert into teacherinfo values(?,?,?,?,?,?,?,?,?,?)";
    int result = 0;
    DBConnection dbcon = new DBConnection();    //创建数据库连接对象
    Connection con = dbcon.getConnection();         // 和数据库建立连接
    PreparedStatement ps = null;
    try {
        ps = con.prepareStatement(sqlstr);      //创建预编译语句对象
        ps.setString(1, teacherinfo.getTeacherId());
        ps.setString(2, teacherinfo.getTeacherName());
        ps.setString(3, teacherinfo.getTeacherBirthday());
        ps.setString(4, teacherinfo.getTeacherSex());
        ps.setString(5, teacherinfo.getTeacherSchool());
        ps.setString(6, teacherinfo.getTeacherTech());
        ps.setString(7, teacherinfo.getTeacherAddress());
        ps.setString(8, teacherinfo.getTeacherInDate());
        ps.setString(9, teacherinfo.getTeacherDepartment());
        ps.setString(10, teacherinfo.getTeacherDuty());
        result = ps.executeUpdate();
    } catch (SQLException ex) {
        System.out.println("增加教员信息时出错");
    } finally {                                     //关闭数据库连接对象
        try {
            if (ps != null)
                ps.close();
            if (con != null)
                con.close();
        } catch (SQLException ex) {
            //do nothing
        }
    }
    return result;
}
```

其中从数据库中删除某一指定员工信息的代码，定义在一个单独的方法 delTeacher (String teacherid)中。该方法接收一个员工的工号（工号同时也是教师登录教务系统的账号），并从数据库中删除该账号所对应的员工信息。该方法的定义如下所示。

```
//在数据库中删除某一员工信息
public int delTeacher(String teacherid){
    String sqlstr = "delete from teacherinfo where rtrim(teacherid)=?";
    int result = 0;
    DBConnection dbcon = new DBConnection();//创建数据库连接对象
    Connection con = dbcon.getConnection(); // 和数据库建立连接
    PreparedStatement ps = null;
    try {
        ps = con.prepareStatement(sqlstr);   //创建预编译语句对象
        ps.setString(1, teacherid);          //设置预编译语句中的参数值
        result = ps.executeUpdate();//执行 sql 语句对象，并返回一个 int 类型的值
```



```
        } catch (SQLException ex) {  //在执行数据库操作时，要处理可能会发生的异常
            System.out.println("删除教员信息时出错");
        } finally {                              //关闭数据库连接对象
            try {
                if (ps != null)
                    ps.close();
                if (con != null)
                    con.close();
            } catch (SQLException ex) {
                //do nothing
            }
        }
        return result;
    }
}
```

27.6　软件部署

至此，惠文教育教务管理系统的登录模块、密码修改模块、人事管理模块等已基本实现。要部署给用户使用，还需要做如下一些工作。

1．实现数据库，导入数据信息

在运行程序之前，按照 27.1.2 节中的说明，将数据库搭建起来，然后启动 MySQL 客户端程序，并进入 accp 数据库，使用如下的参考语句导入相关数据信息：

```
--向班级信息表中导入数据
insert into classinfo(classid,classbid,classteacher,classmanager,classtype,
classseme)
        values ('ACCP2006S10617A', '6071041T01', '陈丽娜', '朱梅松', 'T135AM',
'S1');
insert into classinfo(classid,classbid,classteacher,classmanager,classtype,
classseme)
        values ('ACCP2006S10722A', '6072841T01', '史晓伦', '张慧慧', 'T135PM',
'S1');
insert into classinfo(classid,classbid,classteacher,classmanager,classtype,
classseme)
        values ('ACCP2006S20628A', '6062842Y01', '党国涛', '张淑红', 'T246AM',
'S2');
insert into classinfo(classid,classbid,classteacher,classmanager,classtype,
classseme)
        values ('ACCP2006S20814A', '6081082T01', '王海龙', '朱梅松', 'T246PM',
'S2');
insert into classinfo(classid,classbid,classteacher,classmanager,classtype,
classseme)
        values ('ACCP2006Y20310A', '6031043T01', '姬晓峰', '孙燕', 'T135AM','Y2');
insert into classinfo(classid,classbid,classteacher,classmanager,classtype,
classseme)
        values ('ACCP2006Y20322A', '6032243W01', '周磊', '杨俊英', 'T135PM','Y2');
--向部门信息表中导入数据
insert into DEPARTMENTINFO values ('hw01', '行政部');
insert into DEPARTMENTINFO values ('hw02', '市场部');
insert into DEPARTMENTINFO values ('hw03', '教质部');
```

```
    insert into DEPARTMENTINFO values ('hw04', '学术部');
    insert into DEPARTMENTINFO values ('hw05', '就业部');
    insert into DEPARTMENTINFO values ('hw06', '财务部');
    insert into DEPARTMENTINFO values ('hw07', '后勤部');
    insert into DEPARTMENTINFO values ('hw08', '人事部');
    --向角色信息表中导入数据
    insert into ROLEINFO values ('A','中心校长'); /** 中心校长 */
    insert into ROLEINFO values ('B','部门经理'); /** 部门经理 */
    insert into ROLEINFO values ('C','人事专员'); /** 人事专员 */
    insert into ROLEINFO values ('D','班主任');    /** 班主任 */
    insert into ROLEINFO values ('E','教务');  /** 教务 */
    insert into ROLEINFO values ('F','就业专员'); /** 就业专员 */
    --向成绩信息表中导入数据
    insert into SCOREINFO values ("30112345674","S1_1",0);
    insert into SCOREINFO values ("30112345675","S1_1",0);
    insert into SCOREINFO values ("30112345676","S1_1",0);
    insert into SCOREINFO values ("30112345677","S1_1",0);
    insert into SCOREINFO values ("30112345678","S1_1",0);
    --向学生信息表中导入测试数据
    insert into STUDENTINFO(stuid,stuname,stunamespell,stuclassid,stuteacherid)
        values ('30112345678', '张小三', 'zhangxiaosan', 'ACCP2006S10617A',
'301A001');
    insert into STUDENTINFO(stuid,stuname,stunamespell,stuclassid,stuteacherid)
        values ('30112345677', '李小四', 'lixiaosi', 'ACCP2006S10722A',
'301A002');
    insert into STUDENTINFO(stuid,stuname,stunamespell,stuclassid,stuteacherid)
        values ('30112345676', '王小五', 'wangxiaowu', 'ACCP2006S20628A',
'301A003');
    insert into STUDENTINFO(stuid,stuname,stunamespell,stuclassid,stuteacherid)
        values ('30112345675', '赵小六', 'zhaoxiaoliu', 'ACCP2006S20814A',
'301A001');
    insert into STUDENTINFO(stuid,stuname,stunamespell,stuclassid,stuteacherid)
        values ('30112345674', '朱小七', 'zhuxiaoqi', 'ACCP2006Y20310A',
'301A002');
    insert into STUDENTINFO(stuid,stuname,stunamespell,stuclassid,stuteacherid)
        values ('30112345673', '刘小八', 'liuxiaoba', 'ACCP2006Y20322A',
'301A003');
    --向科目信息表中导入数据
    insert into SUBJECTINFO values ("S1_1","SQL SERVER 应用开发","S1");
    insert into SUBJECTINFO values ("S1_2","程序逻辑和 C 语言实现","S1");
    insert into SUBJECTINFO values ("S1_3","网页编程","S1");
    insert into SUBJECTINFO values ("S1_4","Java 面向对象程序设计","S1");
    insert into SUBJECTINFO values ("S1_5","项目案例","S1");
    insert into SUBJECTINFO values ("S1_6","出勤率","S1");
    insert into SUBJECTINFO values ("S1_7","课堂表现","S1");
    insert into SUBJECTINFO values ("S1_8","COT 课程","S1");
    insert into SUBJECTINFO values ("S1_9","阶段项目","S1");
    insert into SUBJECTINFO values ("S2_1","SQL SERVER 数据库设计和实现","S2");
    insert into SUBJECTINFO values ("S2_2","Java Web 应用开发","S2");
    insert into SUBJECTINFO values ("S2_3","C#和 WinForm 开发","S2");
    insert into SUBJECTINFO values ("S2_4","项目案例","S2");
    insert into SUBJECTINFO values ("S2_5","出勤率","S2");
```

```
    insert into SUBJECTINFO values ("S2_6","课堂表现","S2");
    insert into SUBJECTINFO values ("S2_7","COT 课程","S2");
    insert into SUBJECTINFO values ("S2_8","阶段项目","S2");
    insert into SUBJECTINFO values ("Y2_1","OOAD+XML+Oracle","Y2");
    insert into SUBJECTINFO values ("Y2_2",".NET 企业级应用开发","Y2");
    insert into SUBJECTINFO values ("Y2_3","J2EE 企业级开发及框架技术","Y2");
    insert into SUBJECTINFO values ("Y2_4","项目案例","Y2");
    insert into SUBJECTINFO values ("Y2_5","出勤率","Y2");
    insert into SUBJECTINFO values ("Y2_6","课堂表现","Y2");
    insert into SUBJECTINFO values ("Y2_7","COT 课程","Y2");
    insert into SUBJECTINFO values ("Y2_8","阶段项目","Y2");
    --向员工信息表中导入测试数据
    insert into TEACHERINFO values
      ('301A001', '陈丽娜', '1978-05-06', '女', '清华大学','计算机科学与技术','河
南郑州','2004-07-08','hw04','学术经理');
    insert into TEACHERINFO values
      ('301A002', '党国涛', '1979-06-07', '男', '北京大学','计算机科学与技术','河
南郑州','2004-09-08','hw04','教员');
    insert into TEACHERINFO values
      ('301A003', '崔朝霞', '1976-08-16', '女', '清华大学','计算机科学与技术','河
南郑州','2003-06-18','hw05','教质经理');
    insert into TEACHERINFO values
      ('301A004', '杨俊英', '1975-05-26', '女', '河南大学','体育运动学','河南郑州
','2006-07-08','hw05','班主任');
    insert into TEACHERINFO values
      ('301A005', '杨泉', '1980-05-23', '男', '郑州大学','计算机科学与技术','河南
郑州','2006-12-08','hw05','就业经理');
    insert into TEACHERINFO values
      ('301A006', '李娜', '1982-11-06', '女', '河南师范大学','经济管理','河南郑州
','2007-01-08','hw05','就业专员');
    insert into TEACHERINFO values
      ('301A007', '王金柱', '1972-03-05', '男', '哈佛大学','人力资源管理','河南郑
州','2003-08-08','hw01','校长');
    insert into TEACHERINFO values
      ('301A008', '朱丽娜', '1982-11-15', '女', '山东大学','行政管理','河南　郑州
','2006-12-08','hw03','教务');
    insert into TEACHERINFO values
      ('301A009', '张莉', '1983-10-25', '女', '复旦大学','人事管理','河南郑州
','2007-02-08','hw08','人事专员');
    --向用户信息表中导入数据
    insert into USERINFO values (1, '301A007', '111111', 'A');
    insert into USERINFO values (2, '301A003', '111111', 'B');
    insert into USERINFO values (3, '301A009', '111111', 'C');
    insert into USERINFO values (4, '301A004', '111111', 'D');
    insert into USERINFO values (5, '301A008', '111111', 'E');
    insert into USERINFO values (6, '301A006', '111111', 'F');
```

2．组织程序所需资源

本系统中，除了包 accpedu 中的 class 文件之外，还用到了一些图像、XYLayout 布局管理器包和 MySQL 的驱动程序包。为了使这些资源组织得更加合理，保证程序的可移植性，将这些资源布局如图 27.33 所示。

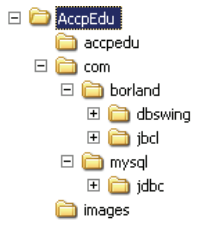


图 27.33　程序资源组织

其中，accpedu 文件夹对应 accpedu 包，内放.class 类文件。com 文件夹下对应两个子目录：borland，内含 XYLayout 布局管理器类；mysql，内含 mysql 驱动程序。images 文件夹下存放应用程序所需的图像。

3．打包程序

在图 27.33 所示的 AccpEdu 文件下，使用记事本创建一个 MANIFEST.MF 文件，文件内容如下。

```
Manifest-Version: 1.0
Created-By: 1.6.0_04 (Sun Microsystems Inc.)
Main-Class: accpedu.AccpApp
```



每一行的冒号与后面的内容之间要有一个空格。每一行结尾要有回车符。

然后，打开命令行窗口（DOS 窗口），进入到如图 27.33 所示的 AccpEdu 目录下，使用如下的命令打包程序文件：

```
jar cvfm accpedu.jar MANIFEST.MF .
```

注意不要遗漏最后面的小圆点（.）。执行以上语句以后，会在当前目录下发现创建了一个 accpedu.jar 文件，如图 27.34 所示。



如图 27.34　可执行 jar 包

4．运行和测试程序

双击图 27.34 中的 accpedu.jar 文件，就会启动教务管理系统的登录界面。按前面几节的功能描述进行测试。这里有几点需要注意：

- 在运行之前，首先要保证数据库服务器处于运行状态。
- 要保证 accpedu.jar 包能双击执行，必须保证应用程序运行所在电脑已经安装了 Java 虚拟机。
- 当移植此应用程序时，保证 com 文件夹、images 文件夹及 accpedu.jar 文件位于同一目录下。

27.7　项目总结

数据库应用程序的开发，是 Java 程序员最经常接触到的项目类型之一。数据库开发技术，同样也是 Java 语言中最重要的技术之一。本案例通过一个真实的惠文教育教务管理系统，着重向读者介绍了从项目背景、需求分析、总体设计和概要设计、数据库设计与实现、功能代码实现到应用程序部署和运行测试的完整的开发过程，比较详细地向读者展示了



Java 数据库应用程序的开发技术。

本项目有如下几个亮点：

- 采用了 MVC 的开发模式，将访问数据库的业务逻辑与前台窗体界面的展示相分离，增强了程序的可扩展性和可维护性。
- 将创建菜单栏和创建数据库连接对象的操作，封装到单独的类中，增强了程序的独立性和灵活性，增加了代码的可复用性，并减少了代码冗余。
- 采用第 4 类 jdbc 驱动程序和开源的 MySQL 数据库，应用程序可移植性高。
- 实用的资源组织和打包实战经验介绍，有助于读者快速掌握可执行程序的创建技术。

当然，作为本书的一个案例，本教务管理系统还有许多有待进一步完善的地方：

- 只实现了登录模块和人事管理模块，读者还可以在此基础上，实现如班级管理、学生考勤、学生成绩管理、学生基本信息管理等功能。
- 在真正开发此类 MIS 系统项目时，一般会借助于一些集成开发环境工具（IDE），如 JBuilder、Eclipse、NetBeans 等，加快开发进度，并有利于进行代码调试和功能测试，提高工作效率。并且使用可视化开发技术，能将程序员从复杂的界面设计中解脱出来。

在学习本章内容时，读者可能会遇到的难点之一是对应用程序业务逻辑的理解，以及 MVC 的开发模式。另外读者在实际调试和运行本程序时，也要注意数据库驱动程序的配置。如果驱动程序配置不正确，可能会使得程序无法正常运行，对此，读者一定要注意。

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888
传　　真：（010）88254397
E-mail：　dbqq@phei.com.cn
通信地址：北京市万寿路 173 信箱
　　电子工业出版社总编办公室
邮　　编：100036